UNIVERSIDAD AUTONOMA DE MADRID

ESCUELA POLITECNICA SUPERIOR

Grado en Ingeniería Informática

TRABAJO FIN DE GRADO

LIBERÍA DE MÉTODOS DE PODA EN CONJUNTOS DE CLASIFICADORES PARA SCIKIT-LEARN

Christian Messina Valverde
Tutor: Gonzalo Martínez Muñoz

Junio 2018

LIBRERÍA DE MÉTODOS DE PODA EN CONJUNTOS DE CLASIFICADORES PARA SCIKIT-LEARN

AUTOR: Christian Messina Valverde
TUTOR: Gonzalo Martínez Muñoz

Dpto. Ingeniería Informática
Escuela Politécnica Superior
Universidad Autónoma de Madrid
Junio de 2018

Resumen

Este Trabajo Fin de Grado consiste en la creación de una librería de ampliación en lenguaje Python para la librería de software de aprendizaje automático scikit-learn. Esta librería implementa un clasificador que usa las técnicas de poda de conjuntos para optimizar el tiempo y la precisión de clasificación de meta-estimadores como Random Forest o Bagging. Todo el código implementado está publicado en un repositorio público de GitHub para su libre uso.

Las técnicas de conjuntos de clasificadores consisten en generar una colección de clasificadores para combinar sus predicciones y dar una decisión única. Estos conjuntos aportan normalmente predicciones más robustas y precisas que un clasificador individual del conjunto. Pero no todo son ventajas: para predecir es necesario llamar a cada clasificador, esto aumenta la capacidad de almacenamiento y de cómputo necesario. Además, no todos los clasificadores de un conjunto predicen igual de bien y es posible que algunos introduzcan error en las decisiones finales. Con las técnicas de poda de conjuntos se propone usar solo un subconjunto del conjunto total de estimadores, de manera que se produzca una reducción en los costes de computación de la predicción, pero con un error de clasificación similar o incluso menor al eliminar los peores clasificadores del conjunto.

Para aplicar estas técnicas se implementa una librería siguiendo la filosofía de diseño de scikit-learn, siendo totalmente compatible con esta. El elemento principal es un clasificador que utiliza algoritmos de poda para un conjunto de estimadores. Existen distintos criterios de poda implementados (reducción de error, medida de complementariedad, etc.) pero también se permite usar un criterio personalizado definido por el usuario. Al entrenar el clasificador este reordenará todos los estimadores según el criterio escogido, y posteriormente aplicará la poda para dejar los estimadores que considere más relevantes. Después de este proceso de poda, se podrá usar el clasificador para la predicción con los métodos habituales de scikit-learn para clasificadores.

En este documento se explicará en detalle todo el proceso de creación de la librería, su motivación, los patrones de diseño usados y más detalles. Además, se incluye la descripción del proceso de pruebas y los resultados de estas, verificando la utilidad de la librería.

Palabras clave

Aprendizaje automático, conjunto de clasificadores, poda de conjuntos, scikit-learn, Python.

Abstract

This Bachelor Thesis consist in the creation of an extension library in Python for the open source machine learning software scikit-learn. This library implements a classifier that uses established pruning techniques to optimize the time and classification accuracy for meta-estimators such as Random Forest or Bagging. All the source code is available in a public GitHub repository for free use.

The ensemble learning techniques consist in generating a collection of classifiers and combine their predictions to give a unique decision. These ensembles usually provide more robust and accurate predictions than an individual classifier of the ensemble. But not everything are advantages: to predict it's necessary to query each classifier, this increases the necessary storage and computation capacities. In addition, not all classifiers from an ensemble have the same prediction accuracy, and it is possible that some of them can introduce more error in the final decisions. The idea behind ensemble pruning techniques is to only use a subensemble from the total ensemble of classifiers, to reduce computational complexity and necessary storage capacity, but maintaining the same accuracy rate or improving it.

To apply these techniques a library is implemented following the design philosophy of scikit-learn and being fully compatible with it. The main element is a classifier that uses ensemble pruning techniques. There are several implemented pruning criteria (reduce error, complementary measure, etc.) but it is also possible to use a personalized criterion defined by the user. When training the classifier, it will rearrange all the estimators according to the chosen criteria. Then it will prune the ensemble to leave the considered more relevant estimators. After this pruning process, the classifier can be used for prediction with the common methods of scikit-learn classifiers.

This document aims to explain in detail the entire process of creating the ensemble pruning library, its motivation, the design patterns used and more details. In addition, the description of the testing process and the results are included, verifying and validating the usefulness of the library.

Keywords

Machine learning, ensemble learning, ensemble pruning, scikit-learn, Python.

Agradecimientos

Doy gracias por haber tenido la oportunidad de vivir esta experiencia. Gracias a los profesores que han sabido transmitirme los conocimientos y valores para ser la persona que soy hoy en día, desde que estaba en infantil hasta la universidad, todos han formado parte de mi formación hasta ahora. Por haber tenido paciencia y por su esfuerzo incansable. Me llevo muy buenos recuerdos de muchos de ellos.

Doy gracias también por mis compañeros, que sin ellos este camino no habría sido ni la mitad de divertido y desde luego mucho más difícil.

Gracias a mi familia que siempre me ha apoyado en las distintas etapas de mi vida, y que siempre han cuidado de mí. Doy gracias por mi hermano Alain, por su compañía y amistad durante todo este tiempo, que son un regalo. Doy gracias en especial a mi madre, porque siempre ha luchado por mi hermano y por mí, porque siempre se ha desvivido para darnos lo mejor, sin ella hoy no estaría escribiendo esto.

Doy gracias por la compañera de batallas y aliada en la vida que tengo sin merecerla. Gracias Lidia por siempre creer en mí y por levantarme, porque sin ti no sería quien soy hoy.

Y, por último, quiero dar gracias a Dios por todo lo que me ha permitido vivir. Ha sido un camino emocionante y lleno de momentos especiales. Estoy expectante por la vida que sigue a partir de ahora.

INDICE DE CONTENIDOS

INDICE DE FIGURAS

INDICE DE TABLAS

1 Introducción

1.1 Motivación

En los últimos años la necesidad de hacer estimaciones, clasificar y predecir información de manera automática ha aumentado de manera sustancial. Las técnicas de aprendizaje automático han dejado de ser herramientas solo del campo científico para introducirse en todos los ámbitos de nuestra vida: redes sociales, motores de búsqueda, diagnósticos médicos, detección de fraude en el uso de tarjetas de crédito, análisis del mercado de valores, reconocimiento del habla, juegos, etc.

Una de las técnicas existentes en el aprendizaje automático es la clasificación. Existen multitud de modelos diseñados para clasificar datos por categorías, sin embargo, una de las técnicas que mejor resultado da es la clasificación por conjuntos de clasificadores [1]. El inconveniente de este método de clasificación es que usar un conjunto grande de clasificadores tiene altos costes de memoria y de computación. De ahí surge la idea de podar estos conjuntos, de manera que se reduzcan estos costes manteniendo la eficiencia [2].

Los algoritmos y las herramientas para el desarrollo del aprendizaje automático no solo avanzan en precisión y eficiencia sino también en ser más accesibles a todo el mundo. Existen librerías en Python conocidas como scikit-learn que reúnen gran cantidad de algoritmos de aprendizaje. La motivación de este TFG es poder contribuir a ampliar la funcionalidad de esta librería añadiendo una librería que implemente métodos de poda de conjuntos de clasificadores.

1.2 Objetivos

El objetivo principal de este trabajo es diseñar e implementar una librería en Python, para la librería de código abierto scikit-learn, que implemente métodos de poda de conjuntos de clasificadores, y por lo tanto siguiendo la filosofía de scikit-learn (tanto de código como de documentación) y siendo totalmente compatible con esta. La librería finalizada se publicará en un repositorio de GitHub público bajo licencia BSD. Con esta librería se pretende conseguir una mejora en el rendimiento y los costes de computación con respecto a los conjuntos de clasificadores enteros, mientras se consigue la misma precisión o incluso una mayor.

1.3 Organización de la memoria

La memoria consta de los siguientes capítulos:

- **Capítulo 1:** Motivación y objetivos del trabajo.
- **Capítulo 2:** Estado del arte. Introducción al aprendizaje automático, a la librería scikit-learn y a los patrones de diseño.
- **Capítulo 3:** Diseño y desarrollo. Descripción de la arquitectura y de los elementos de la librería.
- **Capítulo 4:** Pruebas y resultados de rendimiento, tasa de precisión y costes.
- **Capítulo 5:** Conclusiones y trabajo futuro.

Consta también de un anexo:

- Anexo A: Resultados de las pruebas realizadas

2 Estado del arte

Antes de hablar sobre el diseño y el desarrollo del proyecto, es importante estudiar previamente el estado de los elementos de aprendizaje automático y los patrones de diseño.

2.1 Aprendizaje automático

El aprendizaje automático es un campo de las ciencias de computación que frecuentemente usa técnicas estadísticas para dar a los computadores la "capacidad de aprender" con datos, sin estar programado de manera explícita. También se puede decir que es el campo de estudio científico que se concentra en los algoritmos de inducción y otros algoritmos que se puede decir que "aprenden". Este campo está muy ligado a la computación estadística, que también se centra en la predicción con el uso de computadores [3].

2.1.1 Conceptos básicos

A continuación de describen algunos de los conceptos importantes en el aprendizaje automático [4], [5]:

- **Instancia, ejemplo:** Una instancia es uno de los ejemplos del conjunto de datos (dataset). Por ejemplo, en el caso de que quisiéramos predecir la marca de un coche, cada coche del conjunto sería una instancia.

- **Atributo, característica:** Es cada uno de los factores que describen una de las instancias del conjunto de datos. Para el ejemplo anterior sería cada característica de un coche: nº de puertas, tamaño de ruedas, tracción delantera o trasera, etc.

- **Objetivo:** Es el atributo o característica que se quiere predecir, el objetivo de la predicción. En nuestro ejemplo sería la marca del coche.

- **Ingeniería de factores:** Es el proceso previo a la creación de un modelo. Se analizan y se limpian los datos. Esto se realiza para eliminar los datos que, en lugar de ayudar en la predicción, generan ruido y aumentan en el error. Este proceso es muy importante en el aprendizaje automático y de los más costosos, ya que normalmente hace falta una combinación de conocimiento del entorno (del negocio, industria, etc.) con el conocimiento de la ciencia de datos.

- **Entrenamiento (training):** Es el proceso en el que se introducen los datos de entrada para crear el modelo. Es el proceso básico del aprendizaje automático. Se detectan patrones y se crean las estructuras que se usarán para la predicción.

- **Prueba (test):** Es el proceso donde se introducen los datos de validación (test) que son distintos a los de entrenamiento, para validar el sistema después de la fase de entrenamiento.

- **Modelo:** Es la estructura resultado de entrenar un sistema. La mayoría de los algoritmos inductivos generan modelos que se pueden usar como clasificadores o regresores.

- **Precisión:** Es la tasa de acierto de las predicciones del modelo sobre un conjunto de datos. La precisión se suele estimar usando un conjunto de datos de test independiente que no se haya usado para el proceso de entrenamiento.

- **Aprendizaje supervisado:** Es un proceso de aprendizaje para generalizar problemas cuando se requiere una predicción. Para aprender se compara predicciones del modelo con las respuestas conocidas y de esta manera hace correcciones del modelo.

- **Aprendizaje no supervisado:** Esta técnica de aprendizaje usa un conjunto de datos que no están etiquetados. Busca relaciones y patrones entre los datos para encontrar una estructura o forma de organizarlos. Este tipo de aprendizaje se usa en la minería de datos.

- **Clasificación y regresión:** Ambos son conceptos de aprendizaje supervisado. Un clasificador predice la categoría o etiqueta de un ejemplo, mientras que un regresor predice un número o medida.

2.1.2 Clasificadores y conjuntos de clasificadores

Un clasificador es un algoritmo utilizado para asignar una categoría conocida a un elemento entrante no etiquetado. Por lo tanto, permite ordenar por clases elementos entrantes a partir de la información y las características de estos [6]. El tipo de aprendizaje que se usa para un clasificador es el aprendizaje supervisado.
Un clasificador recibe unos datos de entrenamiento, las instancias de estos datos contendrán distintos atributos que pueden ser categóricos (ej.: “azul”, “verde”, “rojo”), ordinales (ej.: “malo”, “normal”, “bueno”) valores enteros (ej.: número de puertas de un coche) o valores reales (ej.: velocidad, presión). Las categorías que pueden tener las instancias se llaman clases, que es lo que tratará de predecir un clasificador. Para el entrenamiento el sistema compara la predicción con la clase conocida de la instancia y hará la corrección necesaria. Un buen clasificador es uno que puede generalizar bien los datos que recibe, ya que clasificará otros datos distintos. Se dice que hay over-fitting cuando un clasificador predice bien para los datos de entrenamiento, pero mal para otros datos fuera de los de entrenamiento [7]. En las figuras 2-1 y 2-2 se pueden ver ejemplos gráficos de clasificación de distintos modelos.

Algunos modelos de clasificación conocidos son [8]:

- **KNN (vecinos próximos):** Clasifica a partir de la información proporcionada por el conjunto de prototipos, comprobando la cercanía a estos según sus atributos. Mide la distancia con K vecinos.

- **Naïve Bayes:** Es un clasificador probabilístico basado en el teorema de Bayes, asumiendo la independencia entre todos los atributos.

- **Perceptrón:** Este algoritmo es un clasificador binario (solo predice entre 2 clases), además es un tipo de clasificador lineal, es decir, hace sus predicciones basadas en la función de predicción lineal.

- **Red neuronal (perceptrón multi capa):** Usa una serie de capas de neuronas para resolver los problemas que no son separables linealmente (el principal problema del perceptrón simple).

- **Árbol de decisión:** Dado un conjunto de datos se producen esquemas de construcciones lógicas (reglas).

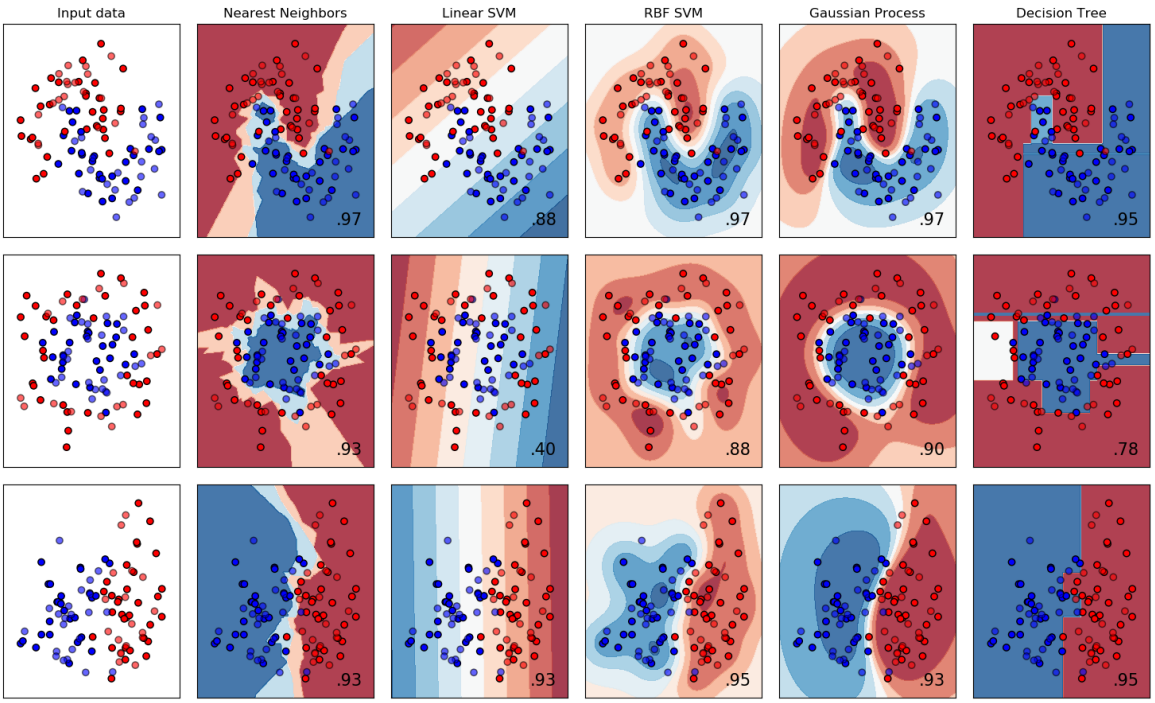

Figura 2-1: Ejemplos de clasificadores clasificando el mismo conjunto de datos

Los conjuntos de clasificadores se usan para conseguir predicciones más precisas y robustas. Consisten en una colección de clasificadores con los que se combina la predicción de estos para dar una decisión única. La predicción de un conjunto de clasificadores suele ser mejor que las de un clasificador del conjunto, ya que al usar múltiples clasificadores se consigue una mejora en la generalización. Algunos ejemplos estimadores formados por un conjunto de clasificadores son:

- **Clasificador Bagging:** Este clasificador entrena sus clasificadores internos (que pueden ser de cualquier tipo) con subconjuntos aleatorios (bootstraps) del conjunto de datos original, esto ayuda a la generalización de la predicción. La predicción del sistema viene de unificar la predicción de todos los clasificadores internos (por votos o por media).

- **Random forest:** Se trata de un conjunto de clasificadores formado por multitud de árboles de decisión. Los árboles se entrenan con subconjuntos del conjunto de datos original, y se usa el promedio para mejorar la precisión y controlar el over-fitting.

- **AdaBoost:** Este algoritmo genera una secuencia de predictores aplicando repetidas veces el mismo algoritmo de aprendizaje a versiones con pesos del conjunto de datos inicial. Al principio todas las instancias tienen un peso uniforme. En cada paso se entrena una copia del clasificador con una versión del conjunto de datos con pesos. Los pesos de las instancias se actualizan en base al rendimiento, los pesos de las instancias mal clasificadas se aumentan mientras que los pesos de las instancias bien clasificadas se reducen. Esto hace que los clasificadores que se vayan generando se centren más en los casos difíciles.

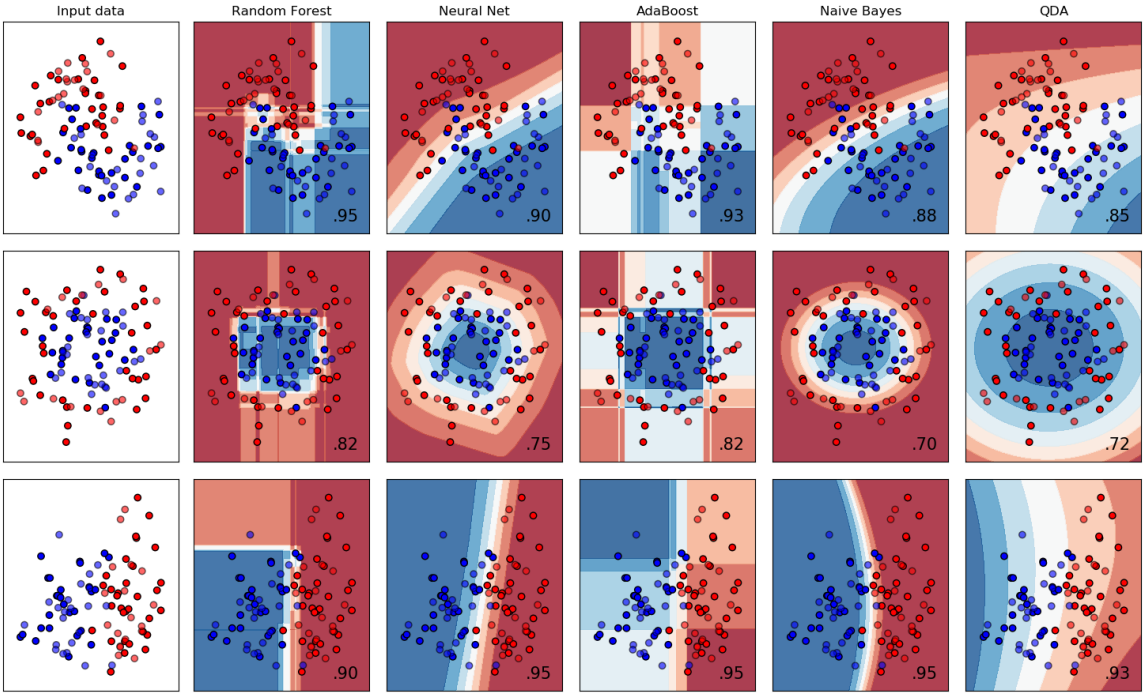

Figura 2-2: Ejemplos de clasificadores clasificando el mismo conjunto de datos

2.1.3 Poda de conjuntos

A pesar de que los conjuntos de clasificadores ofrecen un rendimiento muy destacable, tienen un gran inconveniente, y es que generalmente necesitan combinar un gran número de clasificadores para asegurarse de que el error converge a su valor asintótico. Esto conlleva grandes requisitos de memoria y velocidades de clasificación más lentas. Una forma posible de solucionar estos problemas es escoger solo una fracción de los clasificadores del conjunto original. Además de reducir los costes de memoria y computación, la poda de conjuntos tiene otros beneficios potenciales. Uno de los más importantes es que un subconjunto de los clasificadores originales puede tener un rendimiento mayor en la predicción que el conjunto completo, si se escogen los clasificadores que tengan la mayor generalización.

Escoger los clasificadores que tengan el mayor rendimiento de generalización es un problema complicado con una solución costosa computacionalmente. Si suponemos que para encontrar el mejor subconjunto de clasificadores debemos probar todas las combinaciones posibles de subconjuntos, el coste de la búsqueda crecería de manera exponencial con el número de clasificadores del conjunto original, por lo tanto, calcular de manera exacta el subconjunto óptimo de esta manera es algo irrealizable para los tamaños típicos de conjuntos. Por lo tanto, para superar esto es necesario usar algoritmos aproximados, que con una alta probabilidad seleccionen subconjuntos casi óptimos [2], [9].

Algunos de los algoritmos que se han propuesto para la poda de conjuntos son los algoritmos genéticos, que eliminan el coste exponencial pero aun así suponen un coste computacional bastante elevado. El algoritmo de poda genérico que hemos implementado en nuestra librería consiste en ordenar los clasificadores del conjunto en base a un criterio, para luego definir un punto de poda o corte. A partir de ese punto los clasificadores son desechados y nos quedamos con el subconjunto restante. Los criterios usados son reducción de error [10], complementariedad [11], y UWA [12].

2.2 Scikit-learn

Scikit-learn es una librería de software libre de aprendizaje automático en lenguaje Python. Contiene múltiples utilidades para el tratamiento de conjuntos de datos, algoritmos de clasificación, de clustering, de regresión etc. La última versión estable actualmente es la 0.19.1. La herramienta se puede instalar con el comando:

```
pip install -U scikit-learn
```

O con:

```
conda install scikit-learn
```

Aun que es recomendable instalar una distribución como Anaconda que incluye numpy, scipy, scikit-learn, matplotlib y muchas más utilidades.

La librería scikit-learn contiene bastantes módulos muy útiles, a continuación, hablaremos de algunos de ellos [13]:

- **`datasets`**: Este módulo incluye utilidades para cargar conjuntos de datos desde ficheros que nos permita entrenar y probar los modelos. También contiene distintos métodos para generar datos artificiales, que pueden ser muy útiles para probar distintos escenarios. Además, contiene una serie de conjuntos de datos de "juguete" (toy datasets) que nos permiten hacer pruebas con datos reales no muy grandes. Entre ellos se encuentra el clásico conjunto de datos de iris, el de precio de las casas de Boston, etc.

- **`preprocessing`**: Este módulo contiene varias utilidades para procesar los conjuntos de datos antes de la fase de entrenamiento, de manera que los resultados sean mejores. Incluye métodos de escalado, centrado, normalizado, etc.

- **`metrics`**: Este módulo incluye utilidades para evaluar los resultados generados por los modelos entrenados. Contiene funciones de score, métricas de rendimiento, etc. Algunos métodos importantes son `accuracy_score()`, para medir la precisión de la clasificación de un conjunto de datos (para clasificadores) devuelve un valor de 0 a 1, cuanto más cerca del 1 mayor es la precisión; y `mean_absolute_error()`, para medir la precisión de la predicción sobre un conjunto de datos de un regresor, devuelve el valor de la media del error en positivo, cuanto más cerca de 0 mayor es la precisión.

- **`model_selection`**: Este módulo contiene distintas herramientas para trabajar con un conjunto de datos. Tiene distintos métodos de división de datos como `train_test_split()`. También métodos de validación de modelos, uno de los más importantes y útiles es el de validación cruzada `cross_validate()`.

- **`feature_selection`**: Este módulo implementa algoritmos de selección de atributos. Se puede usar para reducción dimensional de atributos en conjuntos de datos, tanto para aumentar la precisión de un estimador como para aumentar su rendimiento en conjuntos de datos con muchos atributos.

- **`clustering`**: Este módulo reúne algunos algoritmos de clustering de datos no etiquetados (aprendizaje no supervisado). Estos algoritmos se entrenan con los datos no etiquetados y devuelven las etiquetas de los clusters que han creado.

- **`multiclass`**: Este módulo provee una serie de meta-estimadores (estimadores que requieren un estimador base en el constructor). Los algoritmos que incluye permiten usar estimadores binarios o regresores como clasificadores multiclase. También se pueden usar estos algoritmos con estimadores multiclase para intentar mejorar la precisión el rendimiento.

- **`naive_bayes`**: Este módulo implementa algoritmos de Naive Bayes. Son algoritmos de aprendizaje supervisado aplicando el teorema de Bayes, asumiendo la independencia de los atributos.

- **`neighbors`**: Este módulo incluye la funcionalidad para métodos de aprendizaje supervisado y no supervisado basados en vecinos próximos.

- **`neural_network`**: Este módulo incluye modelos basados en redes neuronales.

- **`svm`**: Este módulo incluye algoritmos de máquinas de soporte vectorial (Support Vector Machines, SVMs). Contiene métodos de aprendizaje supervisado que se pueden usar en clasificación y regresión. Son bastante efectivos en espacios dimensionales altos

- **`tree`**: Este módulo contiene modelos basados en arboles de decisión para usar en clasificación y regresión.

- **`ensemble`**: Este módulo incluye métodos basados en conjuntos de estimadores, para clasificación y regresión. El objetivo de estos métodos de conjuntos de estimadores es combinar la predicción de varios estimadores base con un algoritmo de aprendizaje determinado, para mejorar la robustez y la generalización con respecto a un solo estimador. Se suelen distinguir dos familias dentro de los métodos de conjuntos de estimadores [14]:

    - **Métodos de promedio**: Consisten en construir estimadores independientes y luego promediar sus predicciones. Normalmente un estimador combinado es mejor que ninguno de los estimadores base individuales.

    - **Métodos boosting**: En este caso los estimadores base se construyen de manera secuencial, intentando en cada uno nuevo reducir el sesgo del conjunto de estimadores. La idea es combinar varios modelos débiles para crear un conjunto bueno.

- **`utils`**: Este módulo incluye múltiples utilidades muy diversas: métodos de validación de datos, obtención de instancias aleatorias, operaciones de algebra lineal y arrays, operaciones en grafos, funciones de test, funciones de hash, etc.

2.3 Patrones de diseño

En ingeniería de software, un patrón de diseño es una solución general repetible para un problema común recurrente en el diseño de software. No se trata de un diseño terminado que se pueda transformar directamente en código, es una plantilla o una descripción de cómo resolver un problema que puede ser usado en distintas situaciones [15].

Los patrones de diseño ayudan a reducir el tiempo de desarrollo, aportando paradigmas de desarrollos ya probados. En el diseño de software a veces no se consideran algunos problemas que aparecen durante la implementación, al reusar patrones de diseño se ayuda a prevenir estos problemas menores que pueden provocar problemas graves más adelante. Además, se ayuda a mejorar la legibilidad del código para las personas familiarizadas con el patrón [16].

A continuación, se explican los distintos patrones de diseño que hay divididos por categorías:

2.3.1 Patrones de creación

Los patrones de creación son patrones que se encargan de los mecanismos de creación de objetos, intentando crear objetos de una manera adecuada para cada situación. Las formas simples de creación de objetos pueden provocar problemas de diseño, o añadir complejidad a este, estos patrones tratan de solucionar este problema controlando de alguna manera la creación de los objetos [17]. Algunos patrones de creación son:

- **Abstract Factory**. Este patrón trabaja sobre una super factoría que crea otras factorías. Esta interfaz es responsable de crear una factoría de objetos relacionados, pero sin especificar sus clases específicamente.

- **Builder**. Este patrón sirve para separar la construcción del objeto de su representación. A menudo sirve para construir el patrón Composite.

- **Factory Method**. Este patrón crea un objeto sin exponer la lógica de creación al cliente. Es uno de los más usados y provee una de las mejores maneras de crear un objeto. En la Figura 2-3 podemos ver su estructura.

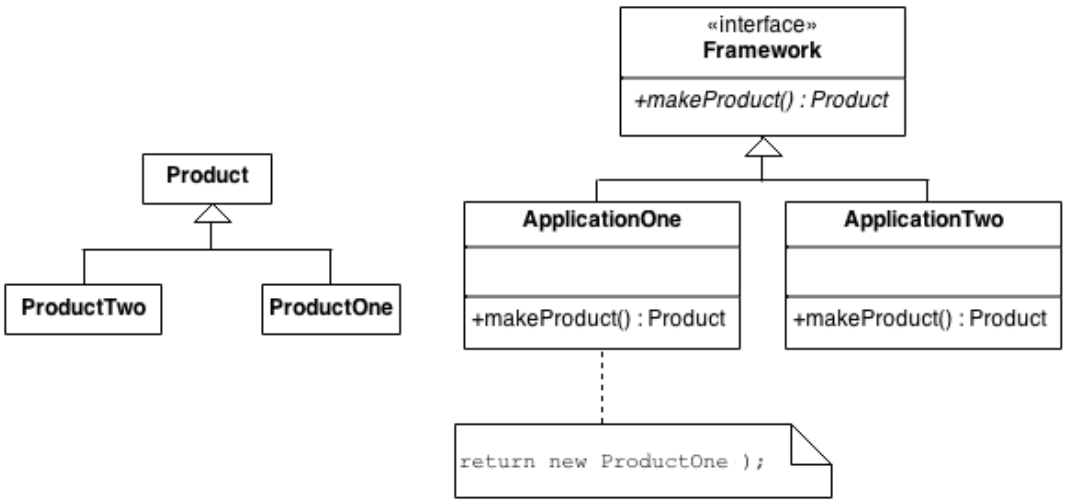

Figura 2-3: Estructura del patrón Factory Method

- **Object Pool**. Este patrón recicla los objetos para evitar el coste de crear y destruir objetos. Es especialmente útil si el coste de iniciar una instancia de una clase es alto y si el número de instancias no es muy alto.

- **Prototype**. Este patrón usa una instancia inicializada para clonar un objeto en vez de crear uno desde cero. Se usa cuando la creación de un objeto directamente es costosa, como por ejemplo cuando implica una operación costosa en una base de datos.

- **Singleton**. Este sencillo patrón sirve para asegurar que solo existe una instancia de una clase determinada. Además, aporta un punto de acceso global a ella.

2.3.2 Patrones de estructura

Los patrones de estructura son patrones de diseño que se encargan de la composición de objetos y clases. Facilitan el diseño identificando una forma simple de crear relaciones entre las entidades [18]. Algunos patrones de estructura son:

- **Adapter**. Este patrón empareja dos interfaces de distintas clases. Funciona como un puente entre dos interfaces incompatibles.

- **Bridge**. Este patrón separa la interfaz de un objeto de su implementación. Se usa cuando se requiere que tanto la interfaz como la implementación puedan variar de manera independiente.

- **Composite**. Este patrón compone objetos en estructura de árbol para representar tanto la jerarquía completa como una parte de ella. Básicamente crea una clase que contiene un grupo de objetos de su misma clase. Esta clase provee métodos para modificar su grupo de objetos. Este patrón se usa cuando se necesita tratar un grupo de objetos de una manera similar a uno de los objetos individuales. Podemos ver su esquema en la figura 2-4.

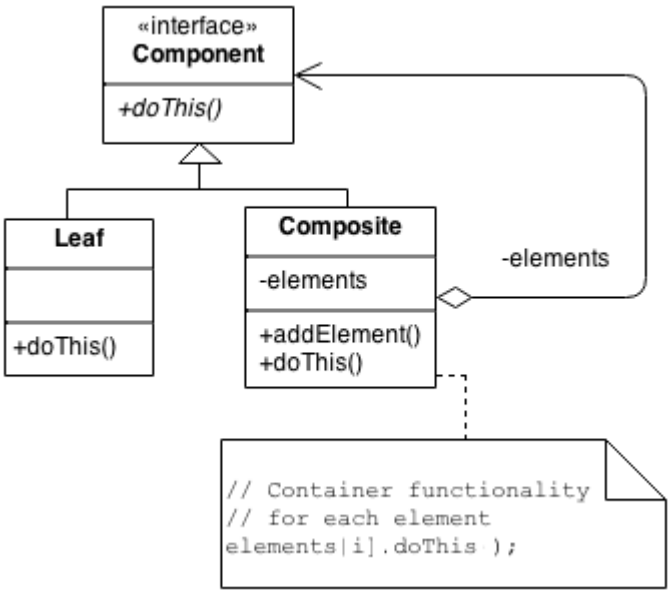

Figura 2-4: Estructura del patrón Composite

- **Decorator**. Este patrón permite añadir funcionalidad a un objeto existente de manera dinámica, sin alterar su estructura. Provee una alternativa flexible a extender la clase para añadir funcionalidad.

- **Facade**. Este patrón provee una interfaz unificada para un sistema. Se crea una única clase con métodos simplificados que llaman a los métodos reales del sistema. Se utiliza para ocultar la complejidad de un sistema aportando una interfaz más simple, haciendo que el sistema sea más sencillo de usar.

- **Flyweight**. Este patrón intenta reusar instancias ya existentes de objetos similares almacenándolos y creando nuevos objetos cuando no se encuentra ningún objeto similar. Se usa para reducir el número de objetos creados, aumentar el rendimiento y reducir el uso de memoria.

- **Proxy**. Este patrón usa una clase para representar la funcionalidad de otra clase. Se crea un objeto proxy que tiene acceso al objeto real. Se usa para proteger el componente real, para dar soporte distribuido y un acceso más inteligente.

2.3.3 Patrones de comportamiento

Los patrones de comportamiento son patrones de diseño que se encargan de la comunicación entre objetos. Identifican patrones comunes de comunicación entre objetos y realizan estos patrones de una manera que aumente la flexibilidad en la comunicación [19]. Algunos patrones de comportamiento son:

- **Chain of responsibility**. Este patrón se usa para enviar una petición a través de una cadena de objetos. Se encadenan los objetos receptores y se les pasan las peticiones que van por la cadena hasta que un objeto la procesa. Esto se hace para mandar la petición a un solo sitio y dejar que procese la petición el objeto adecuado. En la figura 2-5 podemos ver su esquema.

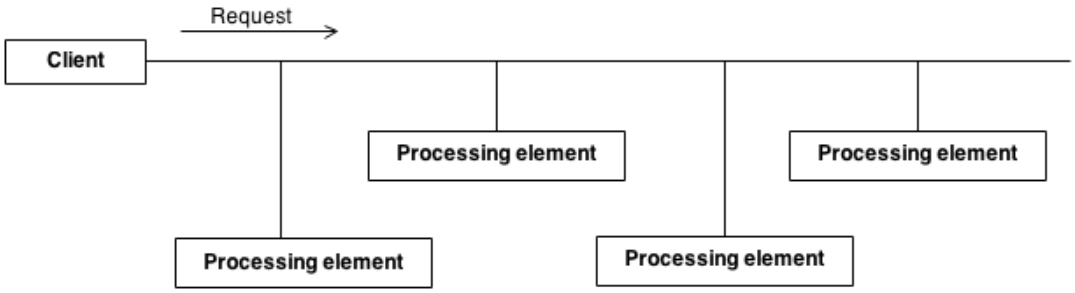

Figura 2-5: Estructura del patrón Chain of responsibility

- **Command**. Este patrón sirve para encapsular una petición en un objeto. Se manda la petición encapsulada al objeto invocador como un comando, este objeto comprueba cual es el objeto apropiado que puede llevar a cabo el comando y le pasa el comando para que lo ejecute.

- **Interpreter**. Este patrón se utiliza para incluir elementos del lenguaje en un programa. Se usa en el parseo de comandos SQL, en el procesado de símbolos, etc.

- **Iterator**. Este patrón sirve para tener acceso a los elementos de una colección de manera secuencial, pero sin necesidad de conocer su representación interna (evitando exponerla). Este patrón es muy común en lenguajes como Java.

- **Mediator**. Este patrón se usa para reducir la complejidad en la comunicación entre múltiples objetos o clases. Se crea una clase mediadora que normalmente maneja todas las comunicaciones entre distintas clases.

- **Memento**. Este patrón se utiliza para restaurar el estado de un objeto a un estadio previo. Captura y externaliza el estado interno de un objeto para que se pueda restaurar más tarde sin violar la encapsulación.

- **Null Object**. Este patrón usa una clase de objeto nulo que se puede usar para proporcionar un comportamiento por defecto si los datos no están disponibles. Además, en lugar de comprobar si algún valor es igual a NULL, el objeto nulo refleja una relación en la que no hace nada.

- **Observer**. Este patrón se utiliza cuando hay una relación 1 a n elementos de manera que cuando un objeto es modificado se necesita notificar automáticamente al resto de objetos.

- **State**. Este patrón sirve para cambiar el comportamiento de una clase según su estado. Cuando el estado interno del objeto cambie se modificará su comportamiento, pareciendo que cambia de clase. Usa la filosofía de una máquina de estados.

- **Strategy**. Este patrón usa varios objetos que representan distintas estrategias, y un objeto de contexto en el que su comportamiento varía según su objeto de estrategia. Esto permite modificar e intercambiar el algoritmo de una clase en tiempo de ejecución.

- **Template Method**. Este patrón define en una clase abstracta las plantillas de sus métodos. Las subclases pueden sobrescribir estos métodos con la implementación que necesiten, pero la invocación de estos se hará de la misma manera en cómo está definida en la clase abstracta. Esto permite definir un comportamiento con ciertos pasos, dejando que las subclases reimplementen estos pasos según sus necesidades [20]. En la figura 2-6 vemos la estructura del patrón.

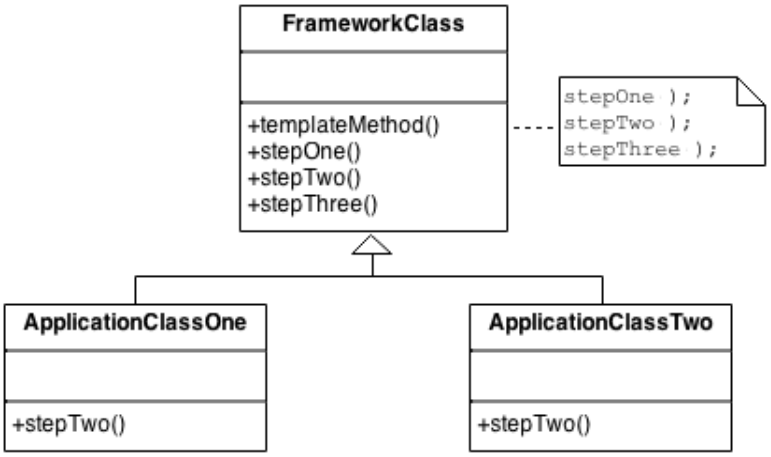

Figura 2-6: Estructura del patrón Template Method

- **Visitor**. Este patrón usa una clase visitante que cambia el algoritmo que ejecuta un elemento de una clase. Esto permite definir una nueva operación sin cambiar las clases de los elementos en los que opera.

3 Diseño y desarrollo

Es esta sección se explican los detalles de la arquitectura, las decisiones de diseño y la implementación del proyecto.

Recordamos que lo que se pretende con este proyecto es desarrollar un software que amplíe la funcionalidad de la librería de Python libre scikit-learn, con una librería que implemente métodos de poda de conjuntos de clasificadores que reducen el tiempo de predicción de los clasificadores de conjuntos y optimizar el uso de memoria. Además, para el diseño del software se tendrá en cuenta la filosofía de diseño de scikit-learn de manera que producto desarrollado encaje en esta filosofía y sirva como un complemento más a la librería scikit-learn.

Todo el código desarrollado para este TFG está al acceso de cualquier persona. Se encuentra en un repositorio público de la plataforma GitHub bajo licencia BSD. A continuación, dejamos el enlace del repositorio: https://github.com/christian5011/ensemble-pruning-sklearn

3.1 Arquitectura y diseño

Para diseñar nuestro proyecto lo primero que tenemos que tener en cuenta es de donde partimos. Como vimos en el capítulo 2, scikit-learn dispone de muchos módulos relacionados con el aprendizaje automático. El módulo que nos interesa principalmente para el desarrollo de nuestro proyecto es el módulo `sklearn.ensemble`, en la Figura 3-1 podemos ver su estructura. Este módulo es el que implementa distintos clasificadores y regresores que utilizan conjuntos de estimadores para aumentar la robustez y la precisión de las estimaciones, ya que se aumenta la generalización.

`sklearn.ensemble`: Ensemble Methods

The `sklearn.ensemble` module includes ensemble-based methods for classification, regression and anomaly detection.

User guide: See the Ensemble methods section for further details.

`ensemble.AdaBoostClassifier` ([...])	An AdaBoost classifier.
`ensemble.AdaBoostRegressor` ([base_estimator, ...])	An AdaBoost regressor.
`ensemble.BaggingClassifier` ([base_estimator, ...])	A Bagging classifier.
`ensemble.BaggingRegressor` ([base_estimator, ...])	A Bagging regressor.
`ensemble.ExtraTreesClassifier` ([...])	An extra-trees classifier.
`ensemble.ExtraTreesRegressor` ([n_estimators, ...])	An extra-trees regressor.
`ensemble.GradientBoostingClassifier` ([loss, ...])	Gradient Boosting for classification.
`ensemble.GradientBoostingRegressor` ([loss, ...])	Gradient Boosting for regression.
`ensemble.IsolationForest` ([n_estimators, ...])	Isolation Forest Algorithm
`ensemble.RandomForestClassifier` ([...])	A random forest classifier.
`ensemble.RandomForestRegressor` ([...])	A random forest regressor.
`ensemble.RandomTreesEmbedding` ([...])	An ensemble of totally random trees.
`ensemble.VotingClassifier` (estimators[, ...])	Soft Voting/Majority Rule classifier for unfitted estimators.

Figura 3-1: Contenido del módulo ensemble de scikit-learn

Después de analizar el contenido del módulo, comprobamos que distingue entre conjuntos de regresores y de clasificadores. Para el objetivo del proyecto nos interesan las clases que implementan clasificadores, que son: AdaBoostClassifier, BaggingClassifier, ExtraTreesClassifier, RandomForestClassifier y VotingClassifier, la clase GradientBoostingClassifier usa una implementación base distinta en su estructura por lo que no se considerará en nuestro proyecto.

Todas estas clases de clasificadores tienen distintas implementaciones, pero comparten dos atributos claves:

- La lista de clasificadores del conjunto que podremos usar para manipular cada clasificador base del conjunto, comprobar sus predicciones, etc.

- Las etiquetas de las clases que nos servirán para la predicción de los clasificadores base del conjunto. Este atributo es útil porque las etiquetas no siempre son números de 0 a n clases.

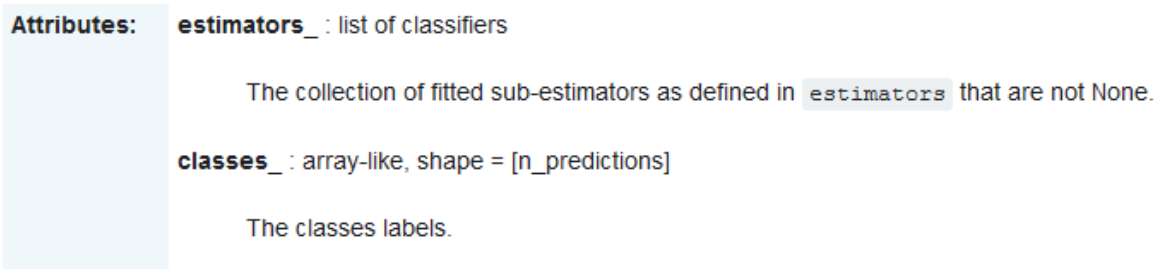

Figura 3-2: Documentación de atributos en común de los clasificadores de conjuntos

Por lo tanto, usaremos estos dos atributos de clasificadores de conjuntos para realizar los distintos algoritmos que usemos en la poda de conjuntos de nuestro software. En la Figura 3-2 podemos ver la documentación de scikit-learn de estos atributos.

El proceso de nuestro algoritmo de poda será el siguiente:

1. Obtener el conjunto inicial de clasificadores ya entrenados.

2. Con un criterio de puntuación determinado, reordenar la lista de clasificadores. Esta reordenación se hace colocando el siguiente clasificador como el que mejor puntuación obtiene con el subconjunto de clasificadores ya ordenados.

3. Determinar un punto de corte en la lista de clasificadores, dejando solo los n primeros clasificadores que serán los que se utilizarán para la predicción.

En el proceso de diseñar la arquitectura de nuestra librería, existía la duda de cómo implementarla siguiendo la filosofía de scikit-learn. Las dos opciones finales eran o implementar métodos de optimización de estos clasificadores, es decir, métodos que modificaran la estructura y los datos de los conjuntos de clasificadores ya implementados por la librería scikit-learn (los vistos anteriormente), u optar por crear nuestro propio clasificador que recibiera un clasificador de conjuntos de clasificadores ya entrenado y que lo usara para aplicar el algoritmo, pero sin modificarlo. Debido a la mayor libertad que nos aporta implementar un clasificador propio además de así evitar modificar la estructura y el

comportamiento de los clasificadores ya implementados (favoreciendo la independencia de estos), nos decantamos por la segunda opción e implementamos nuestro clasificador para scikit-learn: EnsemblePruningClassifier.

En la Figura 3-3 vemos el diagrama de clases en UML de la arquitectura al completo de nuestro sistema, que en las siguientes secciones iremos desgranando para explicar los detalles del sistema y su composición.

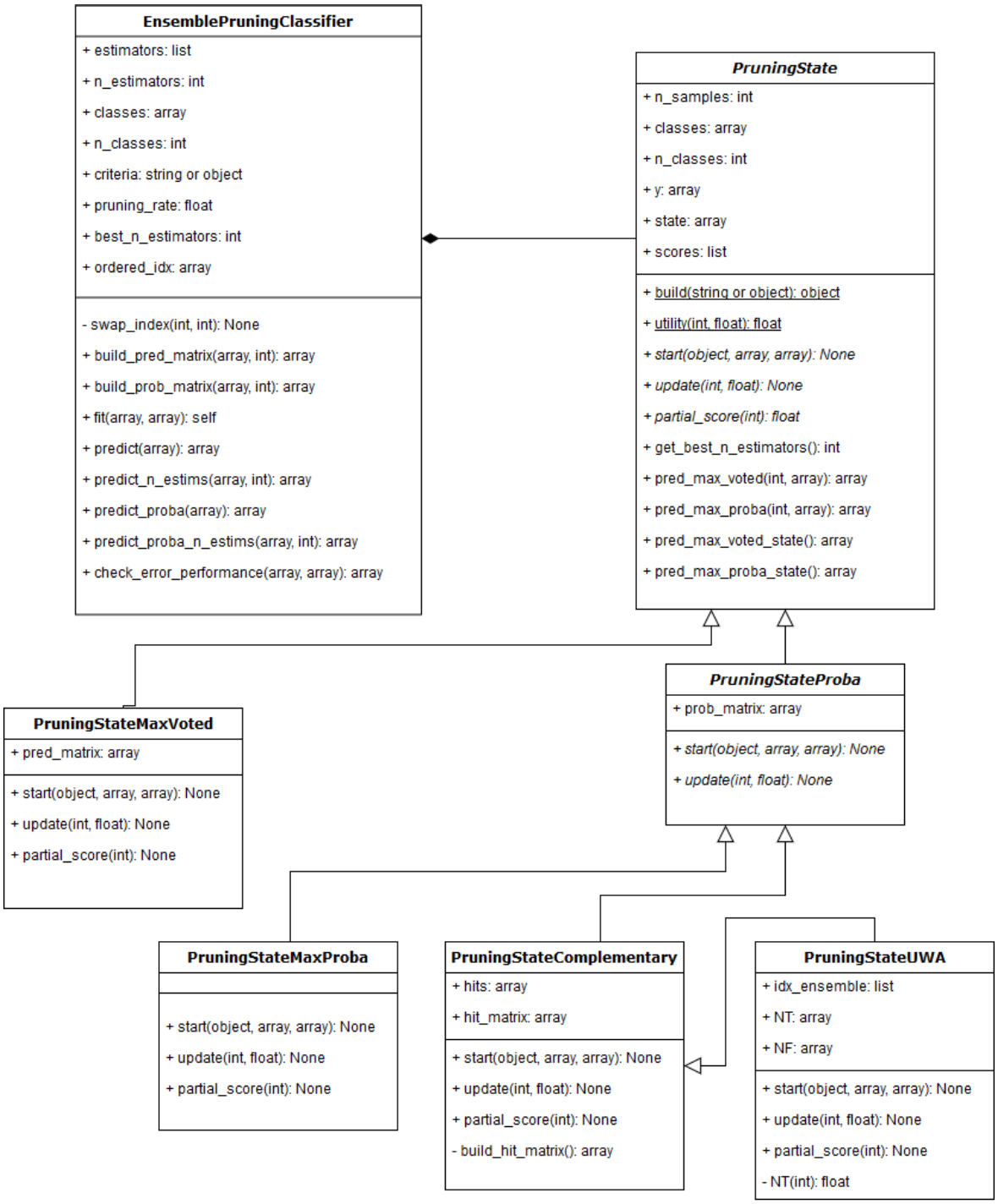

Figura 3-3: Diagrama de clases (UML) del proyecto

3.2 Implementación

3.2.1 EnsemblePruningClassifier

La clase EnsemblePruningClassifier es la clase principal de la librería. Esta clase implementa el clasificador que usa los métodos de poda de conjuntos diseñados. Para implementar un clasificador que encajara con el diseño de scikit-learn es necesario hacer que la clase herede de BaseEstimator que es la clase básica para todos los estimadores, y también de la clase ClassifierMixin que es la clase de la que extienden todos los clasificadores en scikit-learn. Hay que recordar que Python permite implementar herencia múltiple. En la Figura 3-4 se muestra la declaración de clase EnsemblePruningClassifier.

```
class EnsemblePruningClassifier(BaseEstimator, ClassifierMixin):
```

Figura 3-4: Declaración de clase EnsemblePruningClassifier

La clase recibe en el constructor el meta-clasificador de scikit-learn, el criterio con el que se reordenará la lista de estimadores y un parámetro opcional *pruning_rate* (tasa de poda) que permite configurar de manera manual el punto de corte de la poda de estimadores después de reordenarlos, este último parámetro dado con un valor de 0 a 1 (0 significa que se podan todos los clasificadores, 1 significa que no se poda ningún clasificador del conjunto). En caso de dejarlo vacío el punto de poda se realizará de manera automática con un algoritmo que veremos más adelante. En la Figura 3-5 vemos un ejemplo de uso del clasificador de poda de conjuntos. Se puede observar que la librería implementada es muy sencilla de utilizar. En el ejemplo se entrena el clasificador Random Forest y este conjunto se le pasa a EnsemblePruningClassifier junto con el criterio de ordenación, en este caso por mínimo error, y un porcentaje de poda del 20%. Al llamar a *fit* se ejecuta el algoritmo de poda correspondiente sobre el conjunto de random forest que reordena los clasificadores. Tras ello la función *predict* utilizará solo el 20% de los primeros clasificadores para la predicción.

```
rf = RandomForestClassifier(n_estimators=200)
rf.fit(X_train, y_train)

ep = EnsemblePruningClassifier(rf, criteria="max_proba", pruning_rate=0.2)
ep.fit(X_train, y_train)

ep.predict(X_test)
```

Figura 3-5: Ejemplo de uso del clasificador EnsemblePruningClassifier

Al ser un clasificador de scikit-learn hay ciertos métodos que tenemos que implementar, uno de los más importantes es el método *fit*, que se trata del método para la fase de entrenamiento de clasificador. Aquí es donde reside una de las partes más importantes de la lógica que hemos desarrollado. La primera decisión de diseño importante ha sido como gestionar la reordenación de los clasificadores del conjunto original. Para optimizar la reorganización de los clasificadores, en lugar de reordenar la lista directamente, nuestro algoritmo de poda usará una lista de índices, cada valor representa una de las posiciones de la lista de clasificadores original. Estos índices son los valores que de verdad de reordenarán intercambiándose las posiciones en la lista. De esta manera no hace falta crear una copia de la lista de clasificadores que sería una operación muy costosa, y a la vez evitamos modificar

los datos de la estructura original que pertenece al clasificador pasado, favoreciendo el encapsulado.

```
self.ordered_idx_ = list(range(self.n_estimators_))  # Init ordered idx list

state = PruningState.build(self.criteria_)  # Init state
state.start(self, X, y)  # Start state

for i in range(self.n_estimators_):  # Iterate

    scores = []  # Reset the scores
    for j in range(self.n_estimators_ - i):  # Iterate trough the rest of the elements
        # Get the partial score
        score = state.partial_score(self.ordered_idx_[i+j])
        scores.append(score)  # Save the score

    best_score = max(scores)  # Take the best score
    best = scores.index(best_score)  # Take the best score index

    self._swap_index(i, best+i)  # Swap the next index with the best estimator
    state.update(self.ordered_idx_[i], best_score)  # Update state

# Set the best n estimators for prediction
if self.pruning_rate_ is None:  # auto
    self.best_n_estimators_ = state.get_best_n_estimators()
else:  # manual
    self.best_n_estimators_ = int(self.n_estimators_ * self.pruning_rate_)
```

Figura 3-6: Algoritmo de entrenamiento

La siguiente decisión de diseño importante ha consistido en abstraer el criterio de poda del algoritmo. Todos los algoritmos de poda implementados en este TFG siguen un patrón que selecciona el clasificador base que al añadirlo al subconjunto actual mejora en mayor medida un criterio dado.

Como vemos en la Figura 3-6, primero se inicia (o reinicia en caso de que ya se hubiera entrenado con otros datos) la lista de índices (*ordered_idx_*). Después se crea el objeto de la clase *PruningState*, esta clase es el elemento clave para todo el algoritmo y más adelante se explica en detalle. Este objeto se crea antes de la reordenación de los clasificadores (y se elimina al terminar). Se utiliza como estado acumulador del estado actual del entrenamiento. Cumple dos finalidades, optimizar el rendimiento al evitar recalcular muchos valores, y aportar una capa de abstracción para incluir múltiples criterios para la reordenación sin alterar el código del clasificador.

Otro punto importante que comentar antes de introducirnos en el detalle del algoritmo de poda, es el uso de patrones de diseño para la creación de la clase PruningState. Para la creación del objeto usamos el patrón de creación *Factory Method*: el método estático *build* de la clase PruningState recibe como parámetro el criterio de poda, y devuelve el objeto de la subclase de PruningState implementada correspondiente a dicho criterio, sin necesidad de llamar al constructor de la clase adecuada de forma explícita. Esto evita la necesidad de modificar el código del clasificador cada vez que se añada un criterio y hace que el código de poda sea completamente genérico. Además, también usamos el patrón de comportamiento *Template Method*, todas las clases de PruningState comparten los mismos pasos en su ciclo de vida, y llaman a los mismos métodos, pero cada estado lo implementa de acuerdo con sus necesidades.

Después de crear el objeto de estado de poda e inicializarlo, el algoritmo usa un bucle anidado para la reordenación de los índices. El bucle externo recorre todos los clasificadores (índice i). En cada iteración del bucle interno se recorren todos los clasificadores que aún no están ordenados (índice j) y se calcula el score parcial, cuyo cálculo dependerá del criterio. Después de terminar el bucle interno se intercambia el clasificador que dio el mejor score parcial de la iteración del bucle interno con el clasificador en i y se actualiza el estado del objeto PruningState. A continuación, se ilustra este proceso en la figura 3-7.

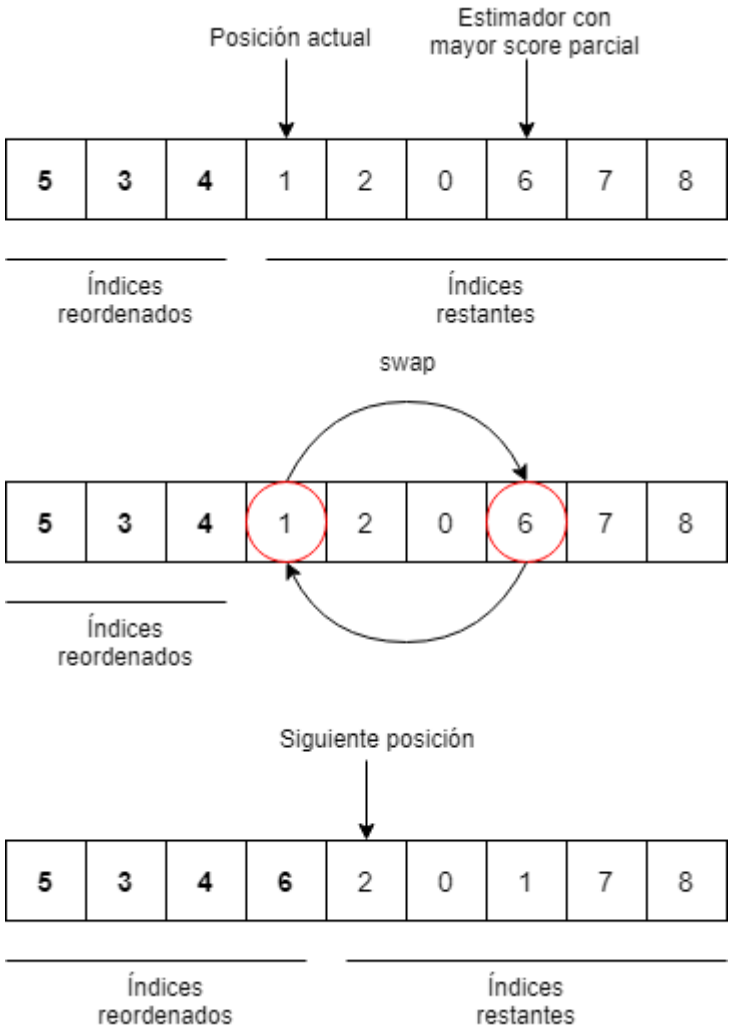

Figura 3-7: Algoritmo de ordenación

Una vez reordenados todos los clasificadores, se selecciona el punto de corte (punto de la poda) de la lista. Si en el constructor del clasificador se pasó el parámetro *pruning_rate*, el punto de corte se establecerá proporcionalmente con el valor dado, con la fórmula siguiente:

$$punto\ de\ corte\ =\ n\ total\ estimadores\ *\ tasa\ de\ poda$$

En caso contrario el punto de corte se establece con el método *get_best_n_estimators*, este método de la clase PruningState devuelve el punto de corte comprobando el score de cada subconjunto posible resultante de hacer un corte en cada posición y devolviendo el valor en el que se obtiene mejor resultado.

Los otros dos métodos implementados comunes a los clasificadores de scikit-learn que implementa EnsemblePruningClassifier son *predict* y *predict_proba*.

- `predict_proba`: Este método común a prácticamente todos los clasificadores de scikit-learn, recibe una colección con las instancias y devuelve la probabilidad de pertenencia a cada clase de cada una de las instancias. Además, nuestro clasificador implementa una variante de predict_proba llamada *predict_proba_n_estims*, que realiza el mismo cálculo que predict_proba pero añade un argumento para indicar cuantos estimadores del subconjunto utilizar para esta predicción. En la llamada a predict_proba estándar el clasificador usará los n clasificadores que haya determinado el algoritmo como número optimo en la fase de entrenamiento.

- `predict`: Este método común a todos los clasificadores recibe una colección de instancias y predice la clase de cada una de ellas devolviendo una lista con las etiquetas correspondientes a la predicción de cada instancia. Nuestro algoritmo de poda además implementa una variante de predict propia, llamada *predict_n_estims* que añade un argumento extra que permite indicar el número de estimadores del conjunto a utilizar en la predicción. El método predict estándar llama a nuestra variante propia usando los estimadores hasta el punto de poda determinado en la fase de entrenamiento. Para determinar la clase que se predice, nuestro clasificador elige la clase que tenga más probabilidad entre todos los estimadores, haciendo una llamada auxiliar a predict_proba_n_estims.

3.2.2 PruningState

La clase PruningState es otro elemento muy importante de la librería que contribuye a su flexibilidad. Como se puede ver en la Figura 3-3, a diferencia de la clase EnsemblePruningClassifier que es una sola clase implementada, la clase PruningState es una clase abstracta que tiene distintas implementaciones. Esta clase se usa para el proceso de poda de clasificadores. La idea de crear una clase que represente el estado de la poda del conjunto viene dada por los siguientes objetivos:

- **Optimización de rendimiento**. Aplicando el algoritmo sin un acumulador el coste en el rendimiento de incrementar el número de clasificadores del conjunto era cúbico, ya que aparte de recorrer en el bucle anidado toda la lista de clasificadores en cada iteración, al llamar a la función que calcula la puntuación parcial con un nuevo clasificador hacía falta volver a predecir con todos los clasificadores del subconjunto ordenado. Podemos comprobar que la predicción de los clasificadores del subconjunto ordenado siempre va a ser la misma en cada iteración del bucle interno ya que son los mismos, por lo que la solución para reducir el coste de cúbico a cuadrático para el número de estimadores es mantener el estado de la predicción del subconjunto, y que cada vez que se añada un clasificador al subconjunto ordenado se actualice este estado.

- **Abstracción**. Otra de las ideas al crear el algoritmo de poda, era permitir usar distintos criterios para la reordenación de la lista de clasificadores sin tener que reimplementar el algoritmo. Esta clase nos permite diseñar una plantilla genérica donde las operaciones básicas del algoritmo están definidas en una clase abstracta externa (patrón de diseño Template Method) para que no sea necesario modificar el código al añadir criterios, ya que todos los criterios comparten la misma plantilla e implementan los mismos pasos. Al usar polimorfismo para los criterios de

ordenación, según cuál sea el criterio definido en el constructor del clasificador, este llamará a los métodos adecuados en el algoritmo de entrenamiento.

Para cumplir los objetivos descritos anteriormente, decidimos usar algunos patrones de diseño de software. A continuación, se explica la aplicación de los patrones de diseño utilizados en la implementación de la clase PruningState:

- **Factory Method**. Este patrón de diseño de creación se usa entre otras cosas para evitar hacer invocaciones específicas a los distintos constructores de la clase PruningState para cada criterio en distintos puntos del código lo que dificulta la ampliación de la librería a nuevos criterios de poda. Se implementa con un método estático llamado *build*. Este método es el que se llama desde el clasificador, pasándole el criterio que se quiera utilizar. El método build comprueba si el criterio es un criterio conocido (cada criterio implementado está ligado a una cadena, esto es en caso de ser una cadena reconocida), en cuyo caso se llama al constructor de la clase concreta adecuada y se devuelve. En caso de que el criterio sea un objeto que extiende de la clase PruningState, lo devuelve directamente como estado de poda para utilizar en el clasificador sin llamar a ningún constructor (ya que el objeto ya está creado). Lo que nos permite esta funcionalidad es dar la capacidad al usuario de crear sus propios criterios de poda, implementado una clase que extienda de PruningState y pasándola como criterio en el constructor de EnsemblePruningClassifier, todo ello sin modificar nada del código de la librería.

- **Template Method**. Este patrón de comportamiento se usa también con la idea de cumplir el objetivo de facilitar la extensión del código a nuevos criterios. Para evitar modificar el código del método fit del clasificador, es necesario establecer unos pasos de comportamiento comunes entre todos objetos PruningState, sea cual sea su criterio de poda (clase concreta). Para ello en la clase abstracta de la que todos los estados de los criterios heredarán, creamos una serie de métodos abstractos que tendrán que implementar todas las subclases. Estos serán los métodos a los que se llame desde el algoritmo de poda. Los métodos que representan los pasos que seguirá cada estado de poda son los siguientes:

    - `start`: Este método se llama al inicio del proceso de entrenamiento del clasificador, para iniciar las variables del objeto estado una vez ya ha sido creado mediante el método estático build (no confundir con el constructor *init*). Este método inicializa todas las variables que necesita el estado antes de empezar el proceso de acumulación. El método abstracto incluye la inicialización de las variables comunes de todas las clases PruningState.

    - `update`: Este método se llama cada vez que se añade un nuevo clasificador al subconjunto de clasificadores ordenados (cada vez que se coloca en su posición el mejor clasificador de los clasificadores no ordenados). Este método se encarga de actualizar las variables del estado necesarias teniendo en cuenta el nuevo clasificador que se añade.

    - `partial_score`: Este método se llama para calcular el score parcial de cada clasificador del resto de clasificadores no ordenados, en conjunto con el subconjunto de clasificadores ordenados.

Además de los métodos abstractos anteriores, la clase abstracta implementa algunos métodos concretos que usan tanto los propios estados como el clasificador:

- `get_best_n_estimators`: Este método se usa al final del algoritmo de entrenamiento del clasificador en caso de que se deje el porcentaje de poda al algoritmo de manera automática. Devuelve el punto de poda en el que la precisión en la predicción sea máxima.

- `pred_max_voted` y `pred_max_voted_state`: Estos métodos sirven para predecir las clases de las instancias por el método de mayor número de votos, es decir, cada clasificador añade un voto a la clase que predice, y la clase que tenga más votos es la clase que escoge el conjunto. El primer método (*pred_max_voted*) recibe un índice de un clasificador por argumentos, lo que hace es predecir con el conjunto formado por el subconjunto de clasificadores ordenados más el clasificador indicado por argumentos. El segundo método (*pred_max_voted_state*) predice con el subconjunto de clasificadores ordenados.

- `pred_max_proba` y `pred_max_proba_state`: Estos métodos predicen las clases de las instancias por el método de mayor probabilidad, es decir, cada clasificador suma sus probabilidades de cada clase por cada instancia, y la clase con la mayor probabilidad es la clase que escoge el conjunto. El primer método (*pred_max_proba*) recibe un índice de un clasificador por argumentos, y predice con el conjunto formado por el subconjunto de clasificadores ordenados más el clasificador indicado por argumentos. El segundo método (*pred_max_proba_state*) predice por máxima probabilidad con el subconjunto de clasificadores ordenados.

Antes de comenzar a hablar de las clases concretas de PruningState que implementan los distintos criterios, hay que comentar que existe otra clase abstracta llamada *PruningStateProba* que sirve de base para todos los criterios que utilicen la predicción por probabilidad, aunque el cálculo del score parcial sea distinto para cada criterio. También se podría haber hecho una clase abstracta como base para todas las clases que utilicen la predicción mediante votación, pero dado que solo hemos implementado una clase que utiliza la predicción por votación no hacía falta. Con esta información cualquier usuario puede implementar su propio criterio de reordenación implementando una clase que herede de PruningState y al menos implementando los métodos abstractos: start, update y partial_score. A continuación, se detallan las implementaciones de los criterios que hemos creado para nuestro clasificador.

3.2.2.1 PruningStateMaxVoted

La clase *PruningStateMaxVoted* implementa un criterio de reordenación que basa el score en la tasa de precisión de la predicción [10], pero esta predicción se da por máxima votación. Es decir, para la predicción del conjunto cada clasificador añade un voto a la clase que predice, y finalmente la clase con más votos es la clase elegida por el conjunto de los clasificadores.

Figura 3-8: Representación de estado de poda PruningStateMaxVoted

Para implementar el estado acumulador con votos, se crea una matriz de nº instancias x nº clases como la de la Figura 3-8, de manera que se van guardando el número de votos de los clasificadores para cada clase. De esta manera, cuando queremos comprobar la predicción del subconjunto actual solo tenemos que ver cuál es la clase con más votos, en lugar de volver a predecir con cada clasificador. Cuando se añade un nuevo clasificador al subconjunto del estado actual, simplemente se añaden sus votos para cada instancia.

3.2.2.2 PruningStateMaxProba

La clase *PruningStateMaxProba* implementa un criterio de reordenación que basa el score en la tasa de precisión de la predicción. La predicción viene dada por la clase con más probabilidad acumulada entre todos los clasificadores.

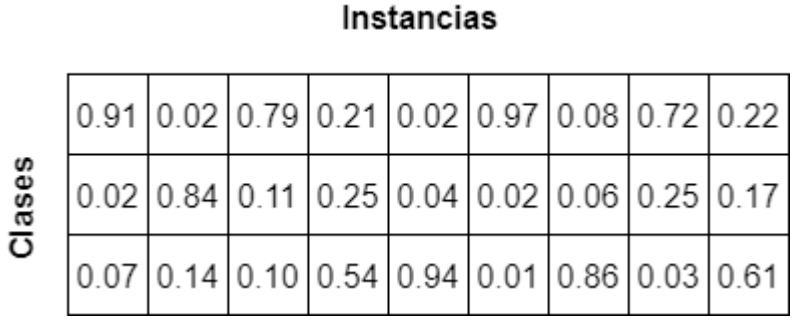

Figura 3-9: Representación de estado de poda PruningStateMaxProba

Al ser una clase que hereda de PruningStateProba, el estado acumulador se implementa con una matriz de nº instancias x nº clases como la de la Figura 3-9. Esto está diseñado en la clase abstracta, todos los estados que hereden de PruningStateProba lo tendrán por defecto.

3.2.2.3 PruningStateComplementary

La clase *PruningStateComplementary* implementa un criterio de reordenación que usa como score la medida de complementariedad [11]. Esta medida calcula para cada clasificador la fracción de ejemplos correctamente clasificados de los que el subconjunto de clasificadores ordenados clasifica incorrectamente. Esto se hace para intentar corregir la predicción del subconjunto al añadir en cada iteración el clasificador que mejor lo complementa, aunque no tiene en cuenta si clasifica correcta o incorrectamente las instancias que el subconjunto clasifica bien. Para optimizar el rendimiento de este criterio se añade una estructura para representar las predicciones correctas e incorrectas tanto del subconjunto actual como de todos los clasificadores del conjunto (ver Figura 3-10).

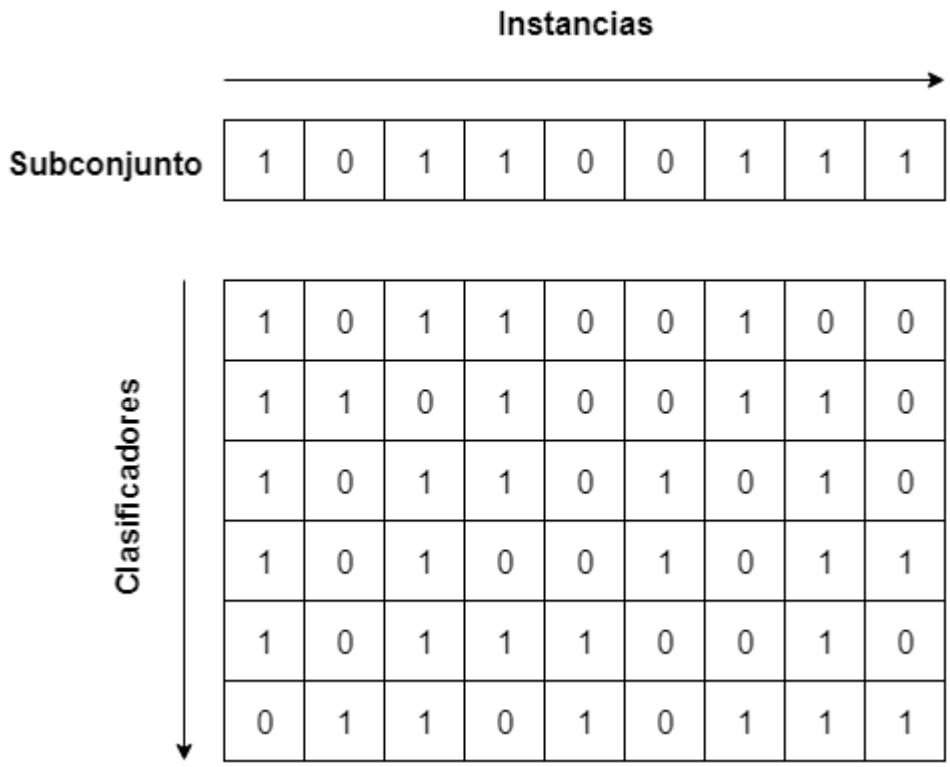

Figura 3-10: Representación estructura de PruningStateComplementary

Se marca con un 1 las instancias que se clasifican correctamente y con un 0 las que se clasifican incorrectamente. La estructura para todos los clasificadores se crea al inicio del entrenamiento con los datos de entrada, y la estructura de predicción para el subconjunto actual se va actualizando cada vez que se añade un nuevo clasificador al subconjunto.

3.2.2.4 PruningStateUWA

La clase *PruningStateUWA* implementa un criterio de reordenación que usa como score el algoritmo *Uncertainty Weighted Accuracy* (UWA) [12]. Este algoritmo comparte la idea de complementariedad, intentando añadir al subconjunto los clasificadores que mejor le complementen, pero realiza cálculos más avanzados que PruningStateComplementary del siguiente modo.

$$\begin{aligned}&\mathrm{UWA}(h_i, \mathcal{S}_{u-1}) = \\ &\quad = \sum_{l=1}^{n} \{\mathbb{I}(e_{tf}(h_i, \mathcal{S}_{u-1}))\mathrm{NT}_l - \mathbb{I}(e_{ft}(h_i, \mathcal{S}_{u-1}))\mathrm{NF}_l \\ &\quad\quad + \mathbb{I}(e_{tt}(h_i, \mathcal{S}_{u-1}))\mathrm{NF}_l - \mathbb{I}(e_{ff}(h_i, \mathcal{S}_{u-1}))\mathrm{NT}_l\} \ .\end{aligned}$$

Esta fórmula indica cómo se calcula la medida de complementariedad para cada clasificador (h_i) junto con el subconjunto ya ordenado (S_{u-1}). El algoritmo itera sobre todas las instancias de entrenamiento y comprueba los 4 casos posibles: que el clasificador clasifique correctamente y el subconjunto incorrectamente, que el clasificador clasifique incorrectamente y el subconjunto correctamente, que ambos clasifiquen correctamente y que

ambos clasifiquen incorrectamente. A continuación, observamos que símbolo de la ecuación se corresponde con cada caso de los anteriores.

$$
\begin{aligned}
e_{tf}(h_i, \mathcal{S}_{u-1}, \mathbf{x}_l, y_l) &: h_i(\mathbf{x}_l) = y_l \wedge s_{u-1}(\mathbf{x}_l) \neq y_l \,, \\
e_{ft}(h_i, \mathcal{S}_{u-1}, \mathbf{x}_l, y_l) &: h_i(\mathbf{x}_l) \neq y_l \wedge s_{u-1}(\mathbf{x}_l) = y_l \,, \\
e_{tt}(h_i, \mathcal{S}_{u-1}, \mathbf{x}_l, y_l) &: h_i(\mathbf{x}_l) = y_l \wedge s_{u-1}(\mathbf{x}_l) = y_l \,, \\
e_{ff}(h_i, \mathcal{S}_{u-1}, \mathbf{x}_l, y_l) &: h_i(\mathbf{x}_l) \neq y_l \wedge s_{u-1}(\mathbf{x}_l) \neq y_l \,,
\end{aligned}
$$

La diferencia fundamental es que este algoritmo tiene en cuenta todos los casos mientras que el algoritmo de complementariedad solo tiene en cuenta el caso en el que el clasificador predice correctamente y el subconjunto incorrectamente. NT_l es la fracción de clasificadores en el subconjunto actual que clasifican la instancia correctamente, y NF_l la fracción de clasificadores que la clasifican incorrectamente ($NF_l = 1 – NT_l$). De esta manera la medida resultante tiene en cuenta como complementa el clasificador al subconjunto de una manera mucho más compleja.

4 Pruebas y resultados

En este capítulo analizaremos las pruebas que se han realizado con la librería desarrollada y los resultados obtenidos. Para las pruebas se han creado diversos scripts que analizan distintos aspectos del rendimiento de nuestro clasificador. Las pruebas realizadas se han dividido en dos categorías principales: pruebas de rendimiento y optimización, y pruebas de coste de ejecución. Las pruebas de rendimiento y optimización analizarán los aspectos relacionados con la tasa de acierto y la poda de los conjuntos. Las pruebas de coste de ejecución analizarán los tiempos del uso de la librería con distintos tamaños de subconjuntos para comprobar el coste cuadrático. Como clasificador base se usa Random Forest.
Todas las pruebas se han ejecutado en un ordenador con las siguientes especificaciones:

- **Procesador:** Intel i7-2600K (4 núcleos y 8 hilos) a 4,5 GHz.
- **Memoria RAM:** DDR3 16GB a 2133 MHz.

Los resultados de todas las pruebas realizadas se encuentran en el Anexo A.

4.1 Conjuntos de datos utilizados

Antes de comenzar con la descripción de las pruebas y los resultados, vamos a describir los conjuntos de datos que hemos utilizado.

Nombre	Descripción	Clases	Atributos	Instancias
Iris	Clasificación de tres especies de flor Iris	3	4	150
Digits	Clasificación de los dígitos decimales	10	64	1797
Wine	Clasificación de tipos de vino	3	13	174
Breast Cancer	Detección de cáncer de pecho	2	30	569
Mushrooms	Clasificación de champiñones entre comestibles y venenosos.	2	112	8124
Phishing	Detección de páginas phishing	2	68	11.055
Skin Segmentation	Clasificación de pixeles según si pertenecen a una zona de piel o no	2	3	245.057

Tabla 1. Conjuntos de datos utilizados

Los conjuntos de datos Iris, Digits, Wine y Breast Cancer provienen de los conjuntos de datos incorporadas en el módulo *datasets* de scikit-learn [21]. Los conjuntos de datos Mushrooms, Phishing y Skin Segmentation se encuentran en la base de datos de LIBSVM [22], y provienen de los conjuntos de datos originales del repositorio UCI [23].

4.2 Pruebas de rendimiento y optimización

Para comprobar el rendimiento que aporta el clasificador entrenado de nuestra librería hemos diseñado algunos scripts que realizan una fase de entrenamiento y una de test con cada criterio implementado. Hemos realizado dos tipos de pruebas en esta sección. Pruebas de

tasa de precisión y pruebas de punto de poda automática. En el Anexo A se incluyen las gráficas de todas las pruebas realizadas.

4.2.1 Tasa de precisión

Para medir la tasa de precisión, se carga el conjunto de datos y después se dividen de manera aleatoria los datos de entrenamiento y de prueba dejando un 70% y un 30% de los datos para cada uno respectivamente. Se crea un clasificador Random Forest con un conjunto de 100 estimadores y se entrena con los datos de entrenamiento anteriores, y después con los mismos datos se poda mediante los 4 criterios de poda descritos con EnsemblePruningClassifier. Después de esto se mide la tasa de precisión (con los datos de pruebas) para cada subconjunto posible usando el punto de poda tanto del conjunto no ordenado (Random Forest), como de los conjuntos reordenados por cada criterio. Este proceso se realiza un total de 10 veces para que los resultados sean más robustos y se calcula la media. También hemos hecho pruebas con distintas medidas de profundidad de los árboles de decisión (3, 5, 7 y sin límite) del clasificador random forest para ver más diferencias.

4.2.1.1 Iris

En la Figura 4-1 podemos ver los resultados de precisión del clasificador para el conjunto de datos *Iris* con profundidad de 3 y sin límite para los árboles de decisión. Podemos ver que para la primera no solo se iguala el resultado de random forest sino que se reduce cuando se usan menos clasificadores. En el estimador 100 (sin aplicar poda) el error de todos los clasificadores tiene que coincidir. Cuando la profundidad de los árboles aumenta, observamos que la reducción del error es inapreciable. Esto creemos que se debe a que cuanto mayor es la profundidad de los árboles, más precisos son individualmente y nuestro algoritmo tiene menos capacidad de optimización del conjunto.

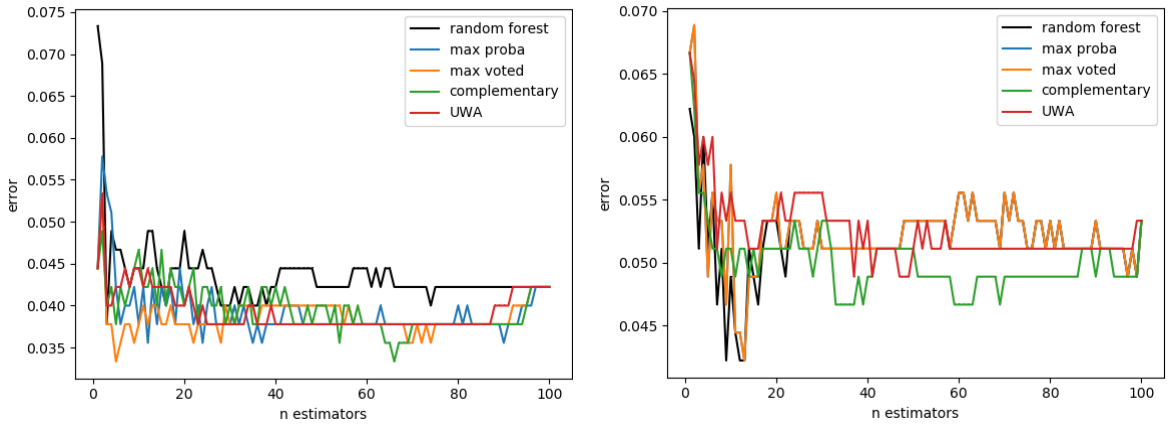

Figura 4-1: Precisión Iris. Profundidad 3 y sin límite

4.2.1.2 Digits

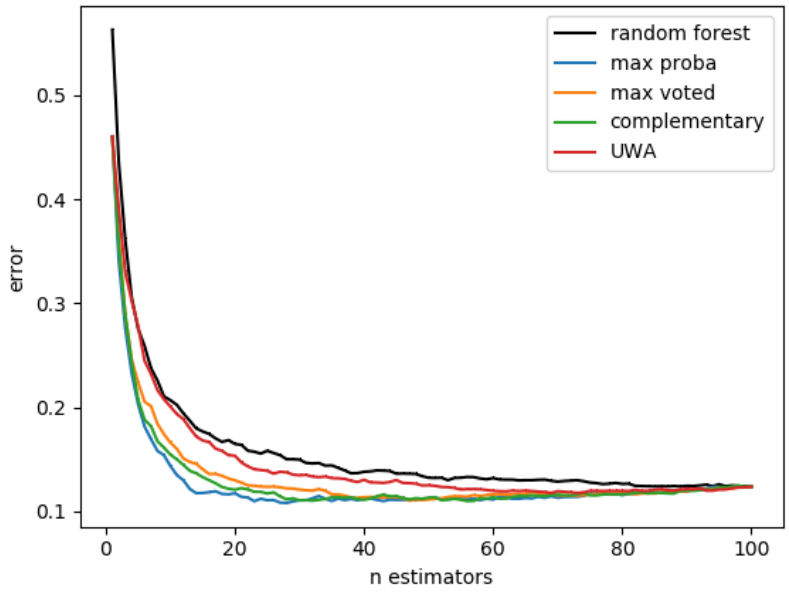

Figura 4-2: Precisión Digits. Profundidad 3

El conjunto de datos *Digits* nos da como resultado unas curvas más limpias que las de Iris. Podemos observar que en la Figura 4-2 para árboles de decisión con profundidad 3, se ve claramente como todos los criterios implementados reducen el error con menos estimadores que random forest y de hecho lo mejoran. El criterio más efectivo en este conjunto de datos es el de máxima probabilidad. En las pruebas con profundidad sin límite en la Figura 4-3 vemos que el margen de mejora es casi inapreciable.

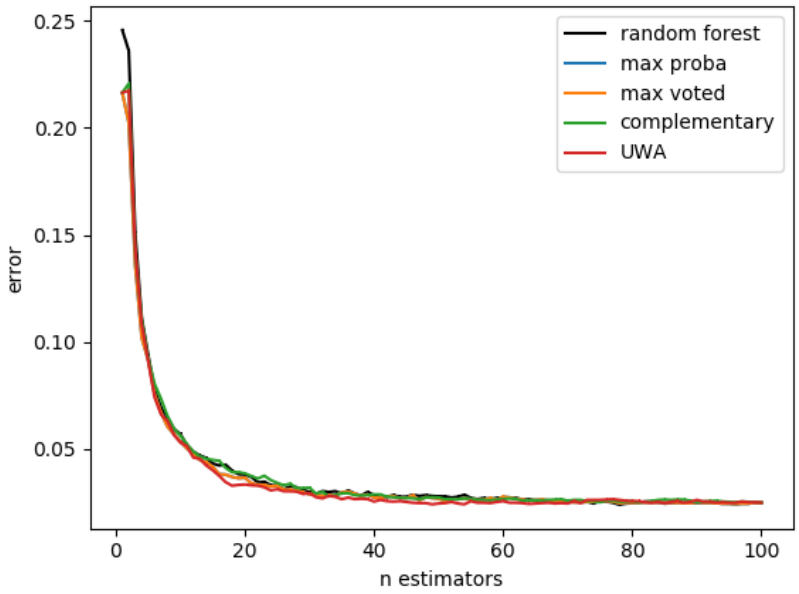

Figura 4-3: Precisión Digits. Profundidad sin límite

4.2.1.3 Wine

Los resultados para el conjunto de datos *Wine* nos muestran que nuestro algoritmo no consigue mejorar los resultados de random forest. En la Figura 4-4 se muestran los datos para profundidad 5 de los árboles. Para el resto de las profundidades los resultados son similares (consultar Anexo A).

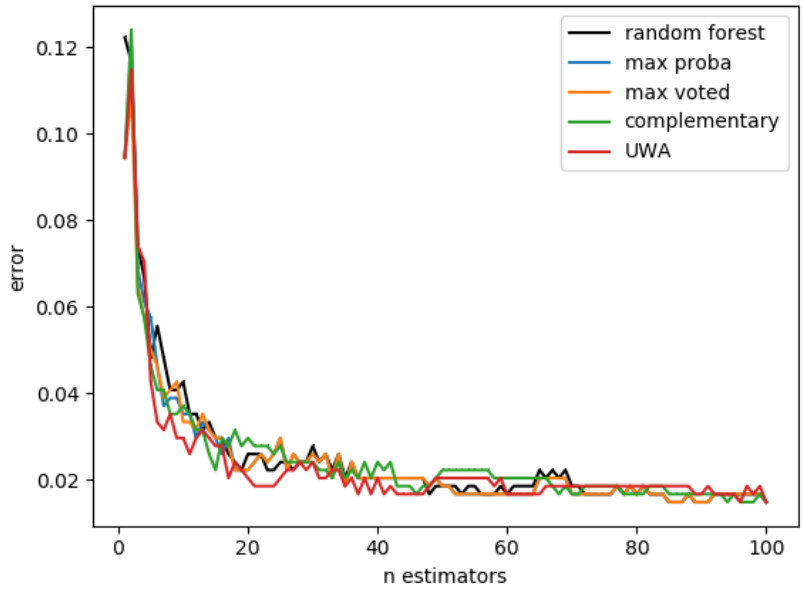

Figura 4-4: Precisión Wine. Profundidad 5

4.2.1.4 Breast cancer

Los resultados para el conjunto de datos de *Breast cancer* son poco aclaradores. En la Figura 4-5 vemos que para profundidad 3 algunos algoritmos sí que mejoran para algunos clasificadores al conjunto sin ordenar, pero dentro del margen de error. Para profundidades mayores no se aprecia mejora.

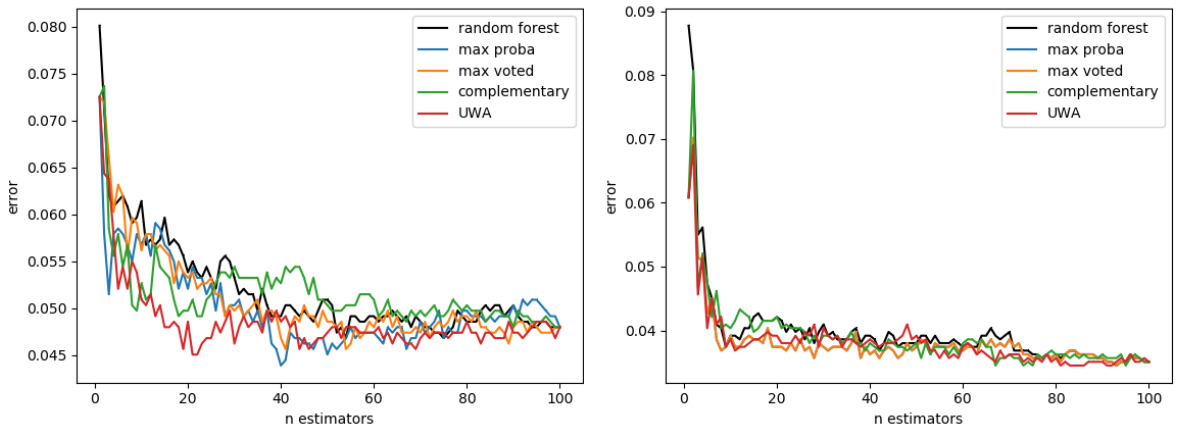

Figura 4-5: Precisión Breast cancer. Profundidad 3 y sin límite

4.2.1.5 Mushrooms

Para el conjunto de datos *Mushrooms* nuestro clasificador da resultados bastante positivos. Para profundidades de árbol bajas la mejora en precisión es bastante alta como se puede ver en la Figura 4-6, donde todos los criterios mejoran sustancialmente al conjunto no ordenado. A medida que sube la profundidad (Figura 4-7) también vemos mejora, pero en este conjunto de datos se llega al límite de máxima precisión en seguida.

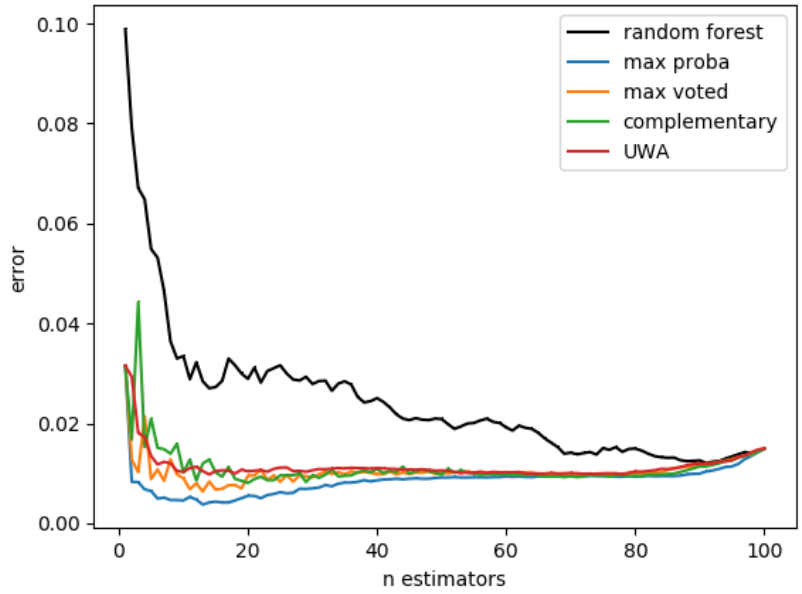

Figura 4-6: Precisión Mushrooms. Profundidad 3

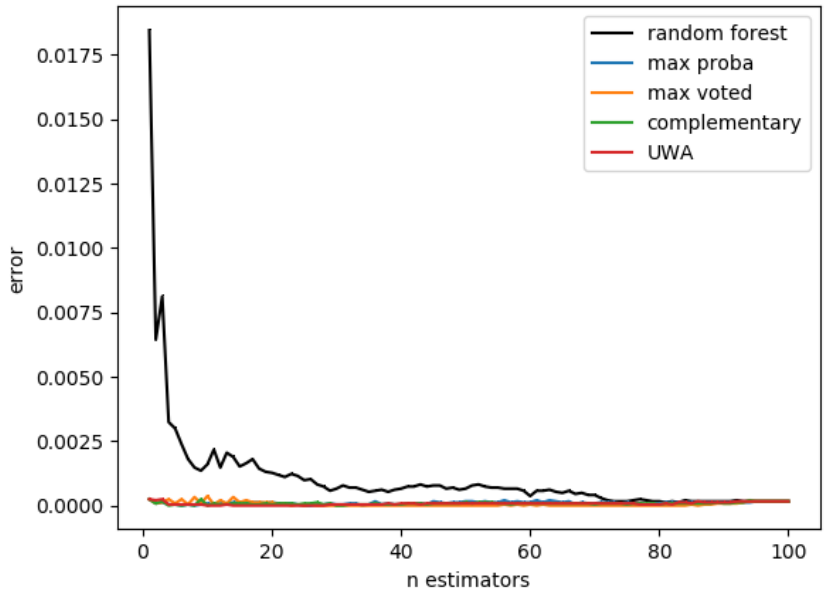

Figura 4-7: Precisión Mushrooms. Profundidad 7

4.2.1.6 Phishing

Para el conjunto de datos *Phishing* los resultados de rendimiento son bastante buenos incluso con una profundidad de 7 que en la que otros conjuntos ya tienen un margen de optimización muy limitado, en la Figura 4-8 podemos ver que sigue existiendo un margen considerable entre el conjunto no ordenado y los conjuntos ordenados de los distintos criterios. El criterio de máxima probabilidad es el que da mejores resultados. Para valores mucho más altos el margen se hace más estrecho (ver Anexo A).

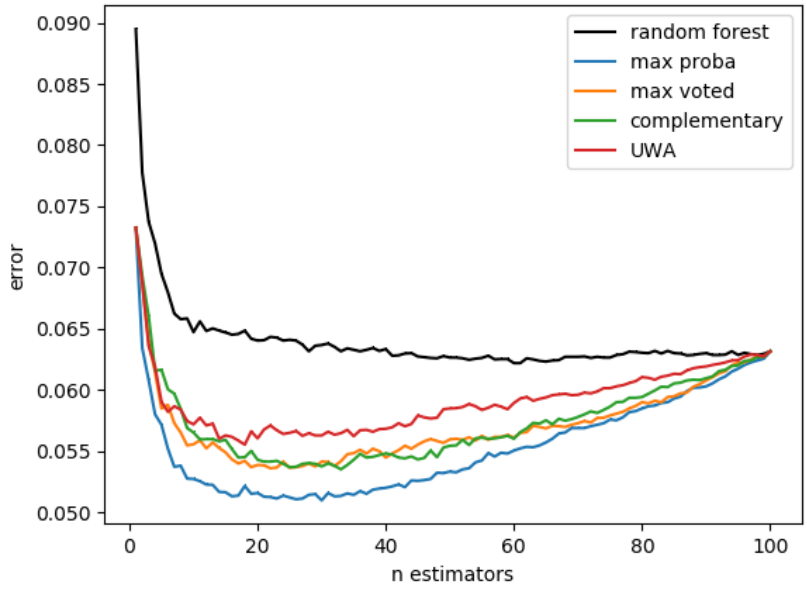

Figura 4-8: Precisión Phishing. Profundidad 7

4.2.1.7 Skin segmentation

Para el conjunto de datos *Skin segmentation*, los resultados son también bastante positivos. Con el criterio de máxima probabilidad por delante del resto, todos los criterios ofrecen una optimización considerable respecto al conjunto no ordenado, incluso con valores de profundidad más altos como en la Figura 4-9. Para valores mucho más altos el margen se reduce (ver Anexo A).

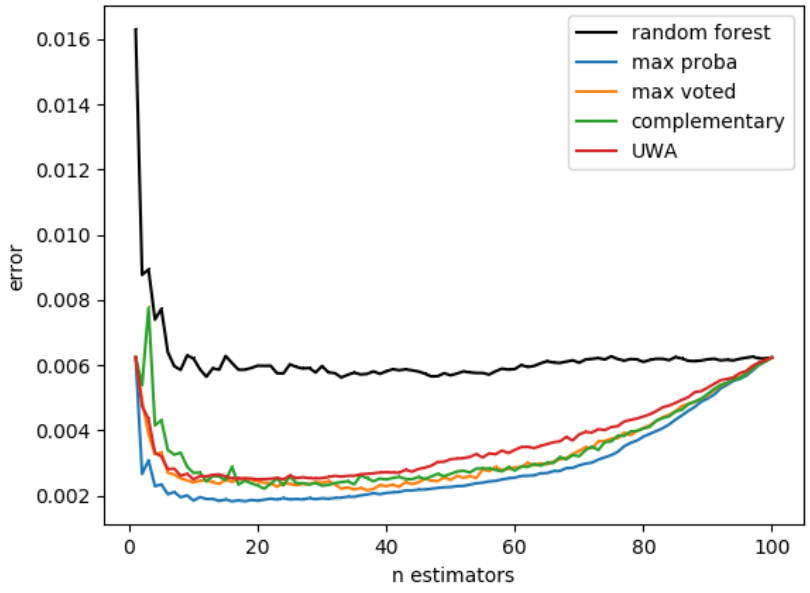

Figura 4-9: Precisión Skin segmentation. Profundidad 7

4.2.2 Tasa de poda óptima

Para las pruebas en las que medimos el punto de poda óptimo para cada conjunto, hemos implementado script que entrena el clasificador de nuestra librería con distintos tamaños para el conjunto de clasificadores (de 10 a 200, con pasos de 10), y calcula el punto de poda con mejor resultado en los datos de entrenamiento para el conjunto. Se hacen 10 repeticiones del cálculo para cada tamaño de conjunto y luego se hace la media, para conseguir resultados más robustos y fiables.

4.2.2.1 Digits

En las pruebas con el conjunto de datos *Digits*, vemos en la Figura 4-10 como a medida que crece el conjunto crece el punto de poda óptimo hasta estabilizarse. Una vez se estabiliza ya no crece con el aumento del conjunto global, esto es porque el algoritmo detecta que el resultado no mejora por añadir más clasificadores. Todos los criterios se comportan de manera parecida, aunque algunos llegan al punto estable antes que otros, y también requieren distintas cantidades de clasificadores para llegar a su punto óptimo.

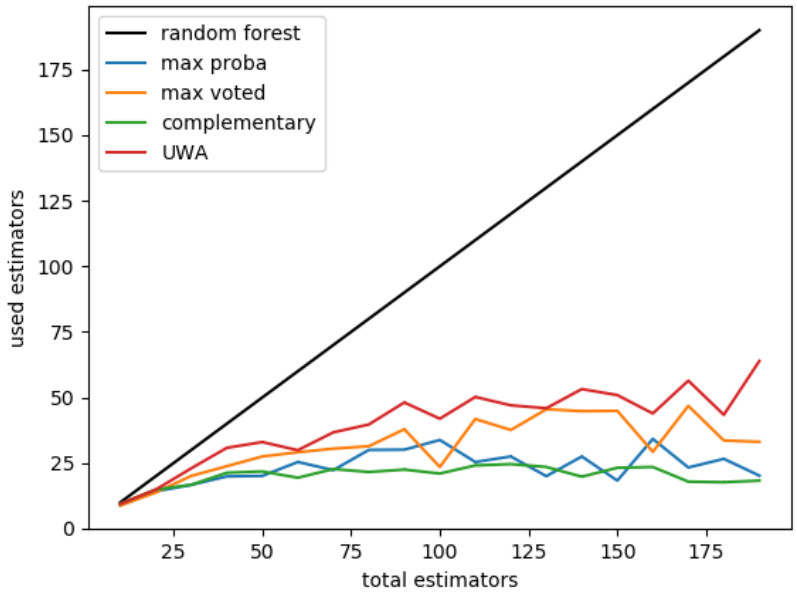

Figura 4-10: Tasa poda Digits. Profundidad 7

4.2.2.2 Phishing

En las pruebas con el conjunto de datos *Phishing* vemos una situación parecida. En este caso vemos que en la Figura 4-11 casi todos los criterios comparten el mismo comportamiento en cuanto al punto de poda, esto es debido a que la tasa de precisión sigue una tendencia parecida en todos los criterios (ver Figura 4-8). En estos gráficos se ve una de las ventajas clave de la poda implementada, y es que es capaz de reducir el conjunto de clasificadores de manera drástica en conjuntos de datos que lo permitan.

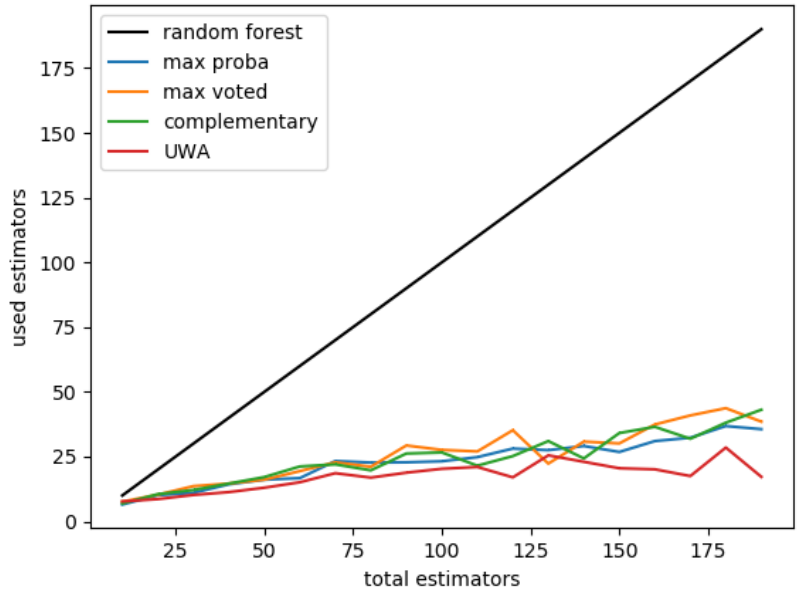

Figura 4-11: Tasa poda Phishing. Profundidad 7

4.3 Pruebas de coste de entrenamiento

Para calcular el coste de entrenamiento de nuestro clasificador hemos implementado un script que entrena el clasificador de nuestra librería con distintos tamaños para el conjunto de poda (de 10 a 200, con pasos de 10), y mide el tiempo de computación de la función fit. Se hacen 10 repeticiones del cálculo para cada tamaño de conjunto y luego se hace la media, para conseguir resultados más estables.

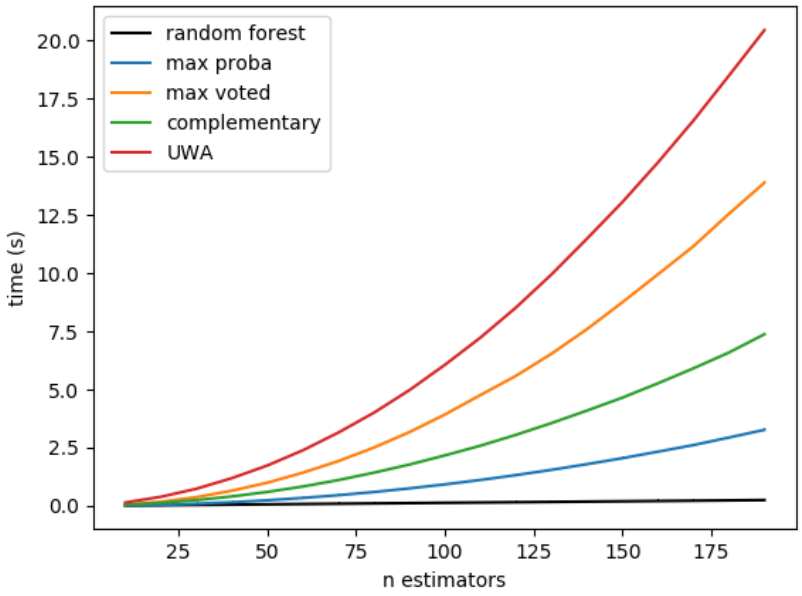

Figura 4-12: Coste entrenamiento Digits. Profundidad 3

En la Figura 4-12 vemos el coste de entrenamiento para el conjunto de datos *Digits*. Aunque cada criterio tiene un coste distinto, todos los criterios tienen un crecimiento de coste cuadrático a medida que aumenta el número de clasificadores. Esto es mucho menor coste al crecimiento exponencial que supondría probar todas las combinaciones de subconjuntos (ver Sección 2.1.3), y también menor que el coste cúbico que supondría realizar el mismo algoritmo, pero sin el uso de un estado acumulador. En el caso del conjunto de datos Digits el criterio de máxima probabilidad es el más eficiente para el entrenamiento.

En la Figura 4-13 vemos la gráfica del coste de entrenamiento para el conjunto de datos *Iris*, como hemos visto antes, todos los criterios tienen un crecimiento de coste cuadrático. Tanto en este ejemplo como en el anterior, vemos que el coste de random forest es prácticamente lineal, esto es porque parte del código de los árboles de decisión de scikit-learn está implementado en C, mientras que nuestra implementación es toda en Python. En este caso el criterio de complementariedad es más eficiente que los demás. Esto nos muestra que no hay un criterio óptimo universal, si no que según las características del conjunto de datos hay criterios que pueden ser más eficientes que otros.

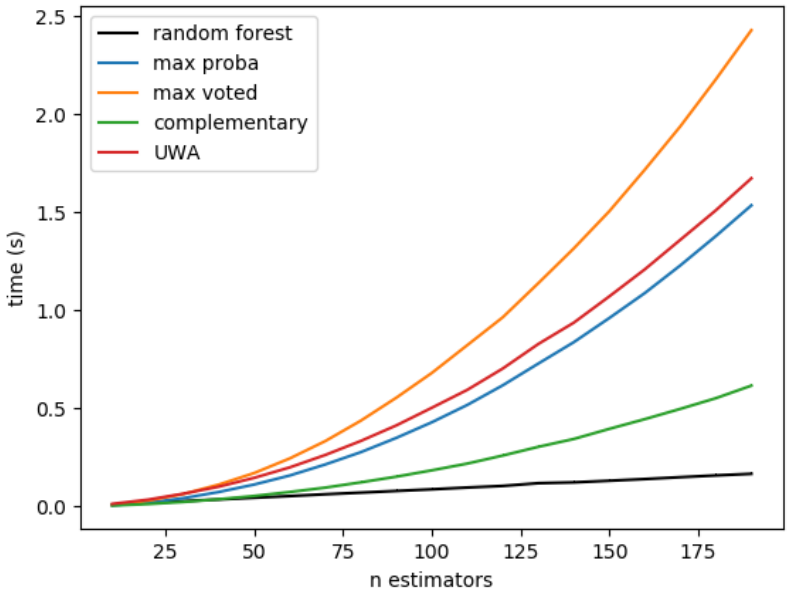

Figura 4-13: Coste entrenamiento Iris. Profundidad 3

La desventaja principal del clasificador de poda de conjuntos frente a un clasificador de conjuntos normal es el coste de entrenamiento. Como hemos visto en las gráficas anteriores el crecimiento del coste de random forest es casi lineal, mientras que la poda de conjuntos tiene un coste cuadrático con el número de clasificadores. Sin embargo, este proceso solo se realiza una vez, y después en la predicción será más rápido que random forest al usar solo una fracción de los clasificadores.

5 Conclusiones y trabajo futuro

5.1 Conclusiones

La creciente demanda de tecnologías de aprendizaje automático debido a su aplicación en múltiples áreas de nuestra sociedad ha producido una rápida expansión en su desarrollo y conocimiento. Una de las aplicaciones más básicas e importantes es la de clasificación de elementos en categorías, realizada por algoritmos clasificadores. Se ha comprobado que el uso de conjuntos de estos clasificadores mejora la robustez y la precisión de las predicciones, sin embargo, supone un coste extra de memoria y computación.

El objetivo de este trabajo era implementar una librería en Python, compatible con la librería de código libre scikit-learn y siguiendo su filosofía de diseño, que implementara métodos de poda de conjuntos. Se optó por implementar un clasificador que dado un conjunto de clasificadores aplicara la poda. Esta poda se realiza ordenando los clasificadores con un criterio dado para después definir un punto de poda. Se han implementado varios criterios de reordenación de los clasificadores, y gracias al uso de patrones de diseño y la abstracción, se deja la posibilidad de implementar más criterios. También se ha implementado un estado acumulador para el proceso de reordenación durante el entrenamiento, de manera que reduzca el coste de computación cúbico con el número de clasificadores a cuadrático.

Los resultados son bastante positivos. Los algoritmos implementados no solo consiguen definir un subconjunto de clasificadores que aporten una tasa de precisión similar, sino que como hemos visto en las pruebas en algunos casos consiguen reducir el error del conjunto entero. Esto nos permite cumplir los objetivos de reducir la complejidad de usar el conjunto sin podar, obteniendo un resultado similar o mejor. Además, como hemos visto en el capítulo 3, toda la implementación se hace con una integración completa con la librería de scikit-learn, cumpliendo otro de los objetivos primarios.

5.2 Trabajo futuro

Como continuación del trabajo para este proyecto existen distintas posibilidades:

- Se pueden desarrollar nuevos criterios de poda que complementen a los ya implementados, que además dada la abstracción del diseño su incorporación al trabajo actual sería bastante sencilla.

- Desarrollar un algoritmo para la selección del punto de poda óptimo teniendo en cuenta más factores que la tasa de precisión máxima.

- Implementar en C el algoritmo de poda o partes de este para reducir el tiempo de computación.

- Otra opción de ampliación sería llevar estos métodos de poda no solo a conjuntos de clasificadores como en este trabajo, sino también a conjuntos de regresores.

Referencias

[1] M. Fernández-Delgado, E. Cernadas, S. Barro y D. Amorim, «Do we Need Hundreds of Classifiers to Solve Real World Classification Problems?,» *Journal of Machine Learning Research,* nº 15, pp. 3133-3181, 2014.

[2] V. Soto, G. Martínez Muñoz, S. García Moratilla, D. Hernández Lobato y A. Suárez, «A Double Pruning Scheme for Boosting Ensembles,» *IEEE Transactions on Systems, Man, and Cybernetics.*

[3] «Machine learning,» Wikipedia, [En línea]. Available: https://en.wikipedia.org/wiki/Machine_learning. [Último acceso: 21 Junio 2018].

[4] A. González, «Conceptos básicos de Machine Learning,» CleverData, 30 Julio 2014. [En línea]. Available: http://cleverdata.io/conceptos-basicos-machine-learning/. [Último acceso: 6 Junio 2018].

[5] J. Brownlee, «Basic Concepts in Machine Learning,» Machine Learning Mastery, 25 Diciembre 2015. [En línea]. Available: https://machinelearningmastery.com/basic-concepts-in-machine-learning/. [Último acceso: 23 Mayo 2018].

[6] «Clasificador (matemáticas),» Wikipedia, [En línea]. Available: https://es.wikipedia.org/wiki/Clasificador_(matem%C3%A1ticas). [Último acceso: 3 Abril 2018].

[7] «Model Fit: Underfitting vs. Overfitting,» Amazon Machine Learning, [En línea]. Available: https://docs.aws.amazon.com/machine-learning/latest/dg/model-fit-underfitting-vs-overfitting.html. [Último acceso: 12 Mayo 2018].

[8] «Statistical classification,» Wikipedia, [En línea]. Available: https://en.wikipedia.org/wiki/Statistical_classification. [Último acceso: 19 Abril 2018].

[9] G. Martínez Muñoz, D. Hernández Lobato y A. Suárez, «An Analysis of Ensemble Pruning Techniques Based on Ordered Aggregation,» *IEEE Transactions on Pattern Analysis and Machine Intelligence,* vol. 31, nº 2, 2009.

[10] D. D. Margineantu y T. G. Dietterich, «Pruning Adaptive Boosting,» *ICML,* pp. 211-218, 1997.

[11] G. Martínez Muñoz y A. Suárez, «Aggregation Ordering in Bagging,» *Int'l Conf. Artificial Intelligence and Applications,* pp. 258-263, 2004.

[12] I. Partalas, G. Tsoumakas y I. Vlahavas, «An ensemble uncertainty aware measure for directed hill climbing ensemble pruning,» *Machine Learning,* vol. 81, nº 3, pp. 257-282, 2010.

[13] «API Reference,» scikit-learn, [En línea]. Available: http://scikit-learn.org/stable/modules/classes.html. [Último acceso: 23 Marzo 2018].

[14] «Ensemble methods,» scikit-learn, [En línea]. Available: http://scikit-learn.org/stable/modules/ensemble.html#ensemble. [Último acceso: 21 Marzo 2018].

[15] «Software design pattern,» Wikipedia, [En línea]. Available: https://en.wikipedia.org/wiki/Software_design_pattern. [Último acceso: 7 Abril 2018].

[16] «Design Patterns,» SourceMaking, [En línea]. Available: https://sourcemaking.com/design_patterns. [Último acceso: 6 Mayo 2018].

[17] «Creational patterns,» SourceMaking, [En línea]. Available: https://sourcemaking.com/design_patterns/creational_patterns. [Último acceso: 23 Mayo 2018].

[18] «Structural patterns,» SourceMaking, [En línea]. Available: https://sourcemaking.com/design_patterns/structural_patterns. [Último acceso: 9 Mayo 2018].

[19] «Behavioral patterns,» SourceMaking, [En línea]. Available: https://sourcemaking.com/design_patterns/behavioral_patterns. [Último acceso: 23 Abril 2018].

[20] «Design Patterns. Template Pattern,» Tutorials Point, [En línea]. Available: https://www.tutorialspoint.com/design_pattern/template_pattern.htm. [Último acceso: 15 Mayo 2018].

[21] «Dataset loading utilities,» scikit-learn, [En línea]. Available: http://scikit-learn.org/stable/datasets/index.html. [Último acceso: 18 Marzo 2018].

[22] «LIBSVM Data: Classification, Regression, and Multi-label,» [En línea]. Available: https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/. [Último acceso: 23 Mayo 2018].

[23] «Machine Learning Repository,» UCI, [En línea]. Available: http://archive.ics.uci.edu/ml/index.php. [Último acceso: 13 Junio 2018].

Glosario

UWA	Uncertainty Weighted Accuracy
UML	Unified Modeling Language

Anexos

A Resultados de las pruebas realizadas

En este anexo se muestran todos los resultados obtenidos en las pruebas realizadas.

Pruebas de tasa de precisión

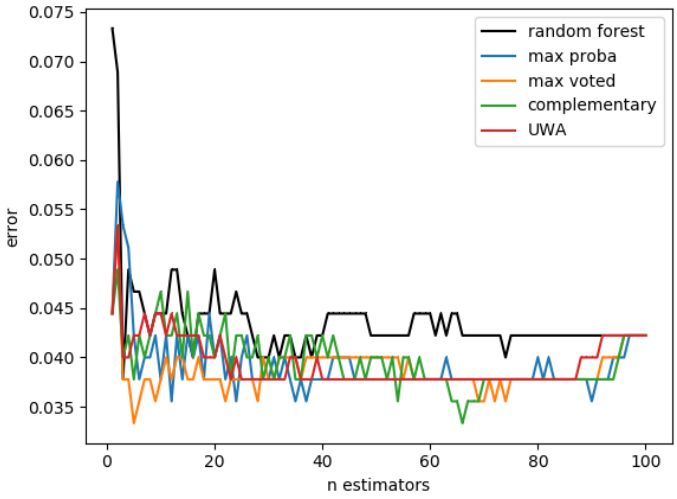

Figura 0-1: Precisión Iris. Profundidad 3

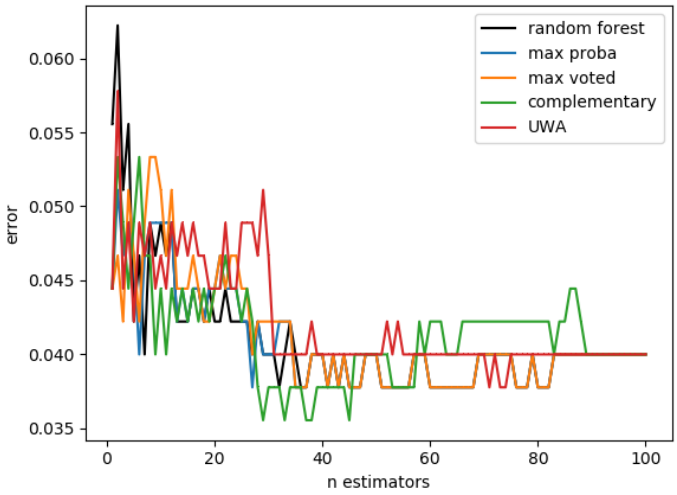

Figura 0-2: Precisión Iris. Profundidad 5

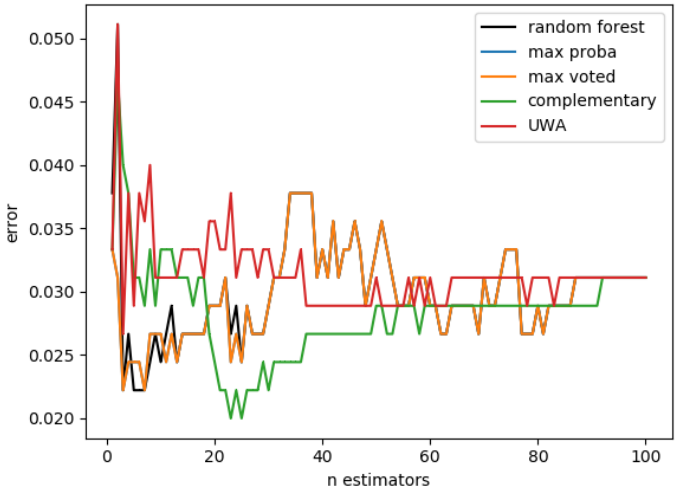

Figura 0-3: Precisión Iris. Profundidad 7

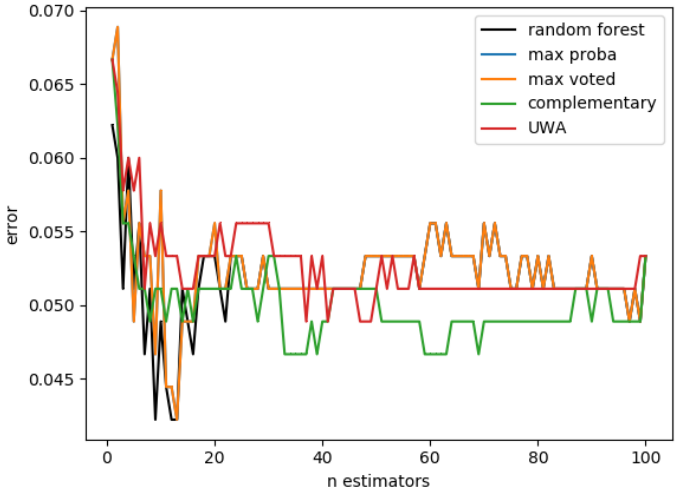

Figura 0-4: Precisión Iris. Profundidad sin límite

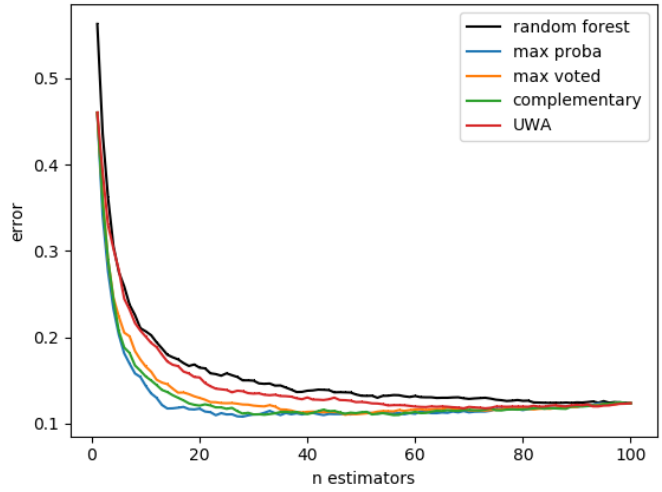

Figura 0-5: Precisión Digits. Profundidad 3

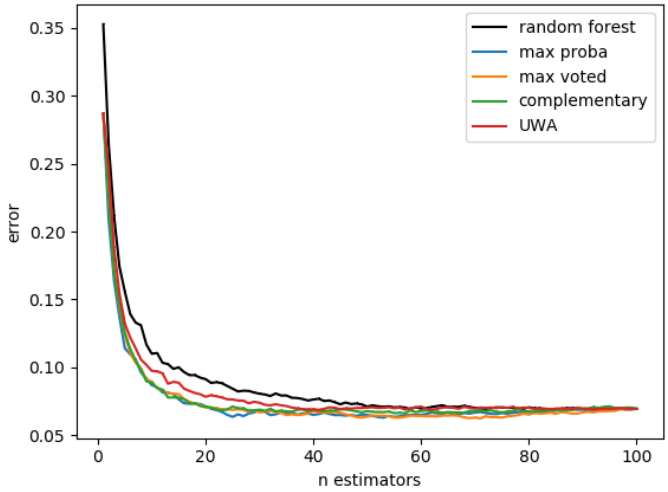

Figura 0-6: Precisión Digits. Profundidad 5

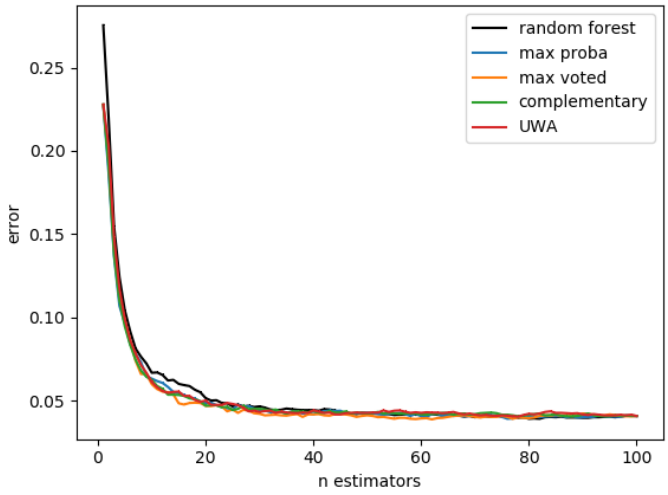

Figura 0-7: Precisión Digits. Profundidad 7

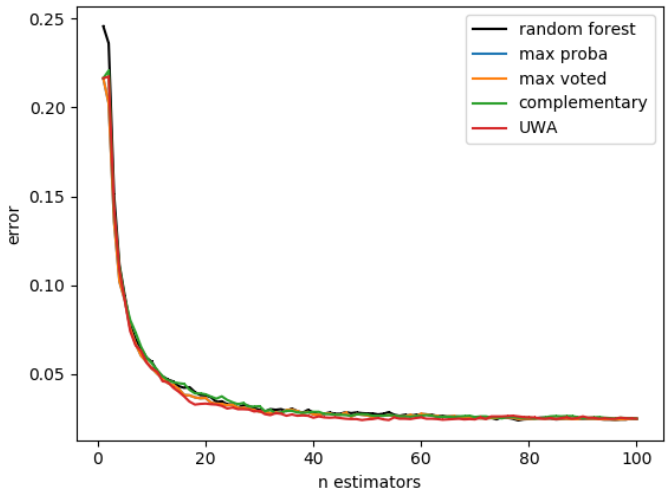

Figura 0-8: Precisión Digits. Profundidad sin límite

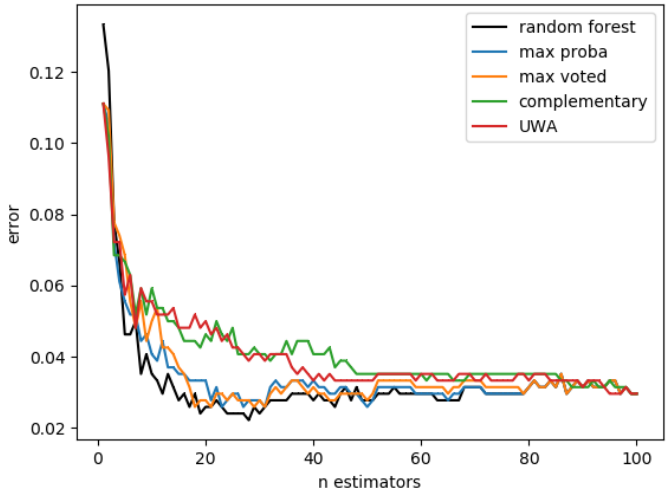

Figura 0-9: Precisión Wine. Profundidad 3

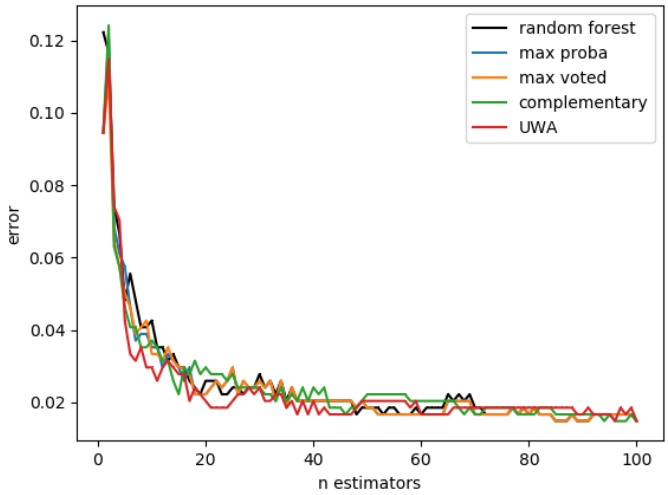

Figura 0-10: Precisión Wine. Profundidad 5

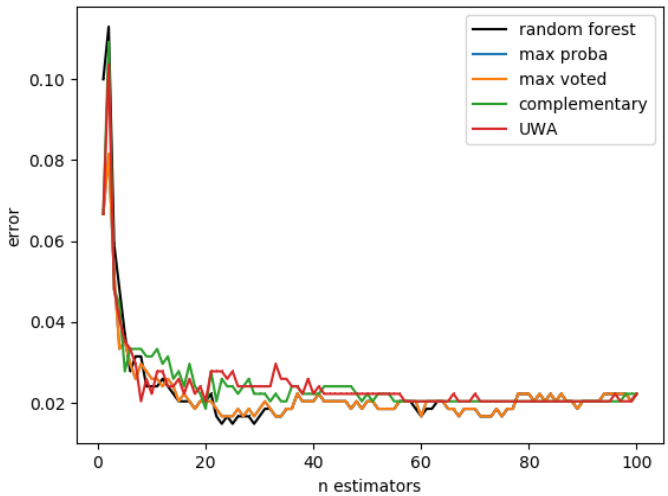

Figura 0-11: Precisión Wine. Profundidad 7

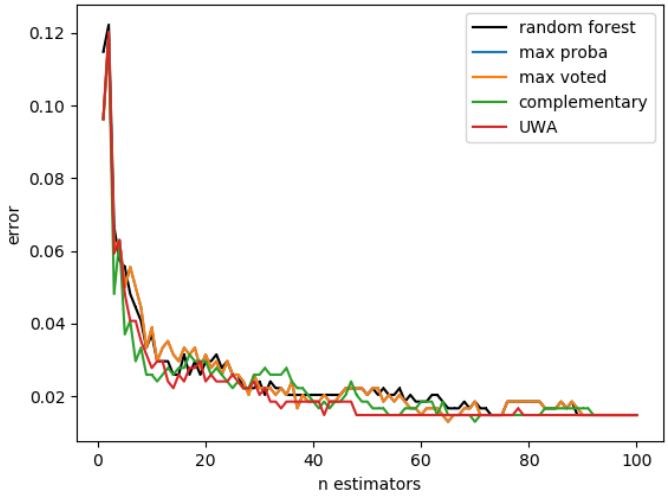

Figura 0-12: Precisión Wine. Profundidad sin límite

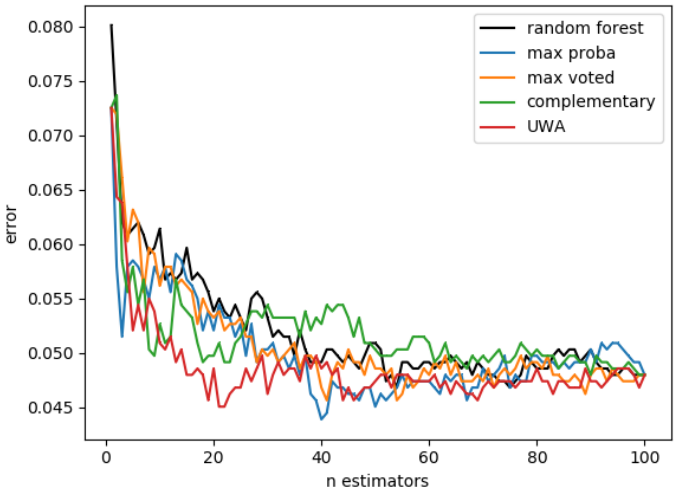

Figura 0-13: Precisión Breast cancer. Profundidad 3

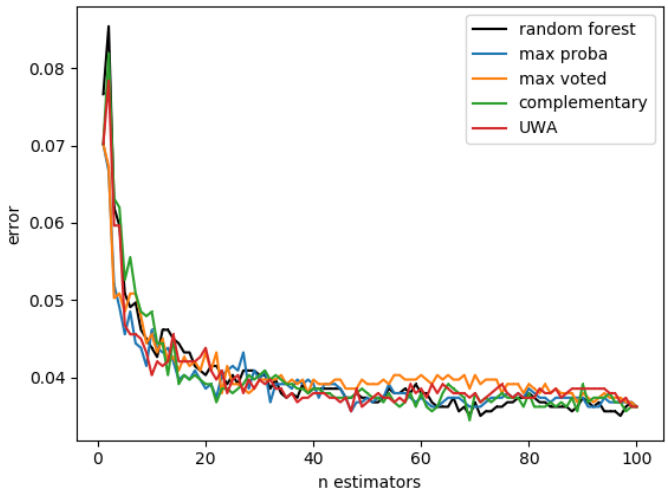

Figura 0-14: Precisión Breast cancer. Profundidad 5

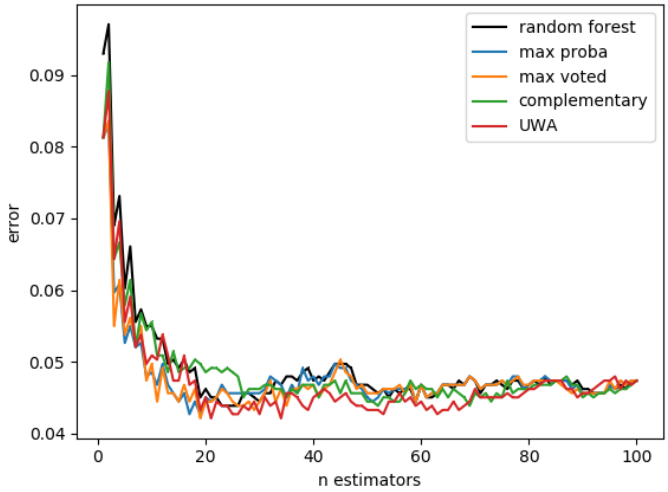

Figura 0-15: Precisión Breast cancer. Profundidad 7

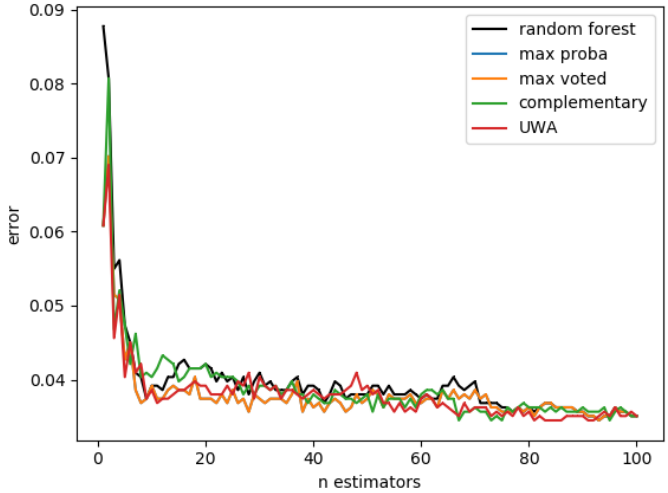

Figura 0-16: Precisión Breast cancer. Profundidad sin límite

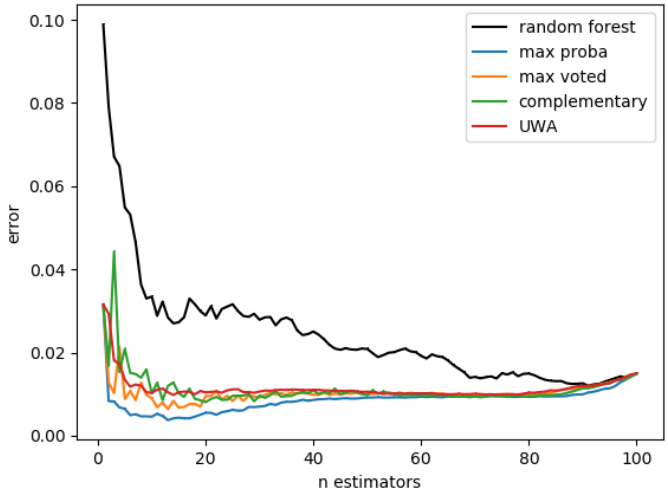

Figura 0-17: Precisión Mushrooms. Profundidad 3

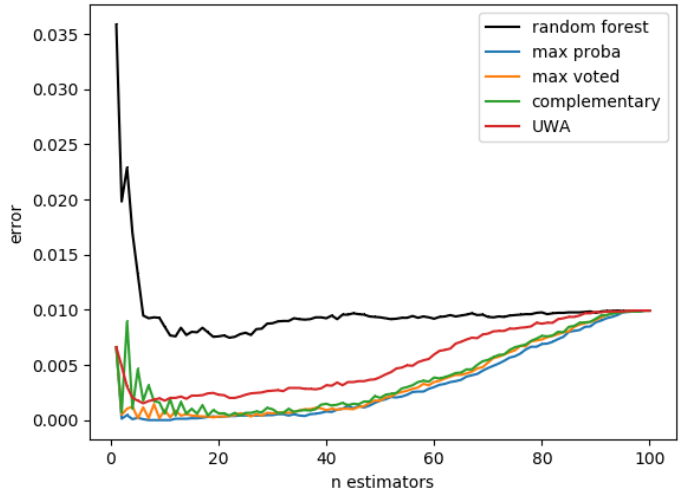

Figura 0-18: Precisión Mushrooms. Profundidad 5

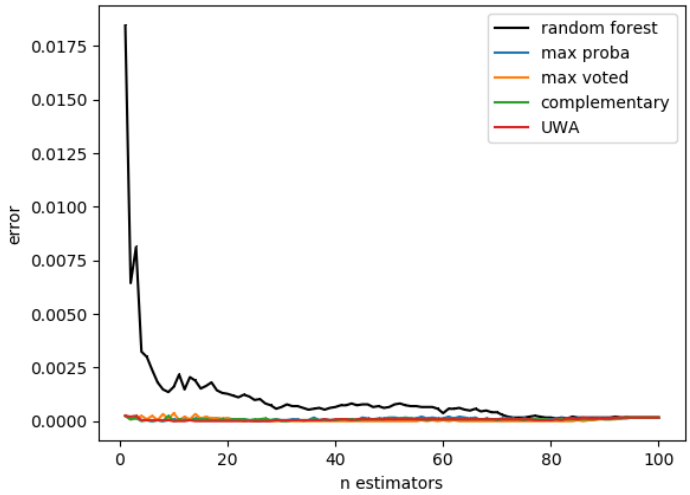

Figura 0-19: Precisión Mushrooms. Profundidad 7

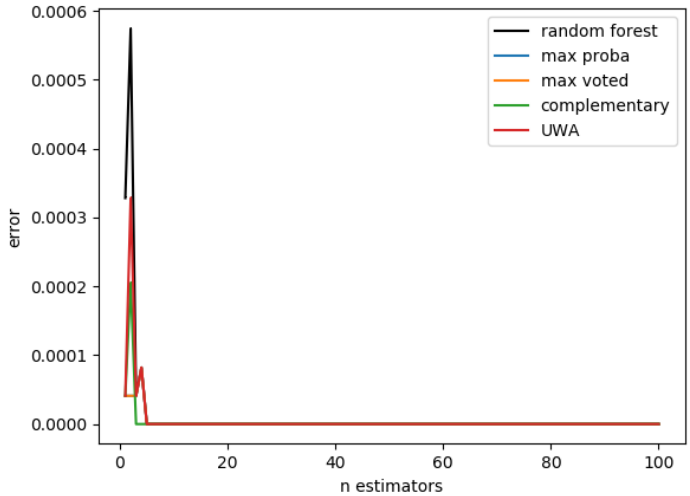

Figura 0-20: Precisión Mushrooms. Profundidad sin límite

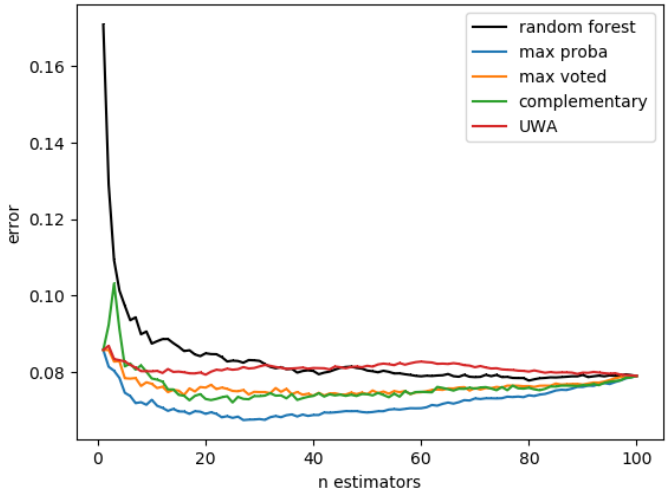

Figura 0-21: Precisión Phishing. Profundidad 3

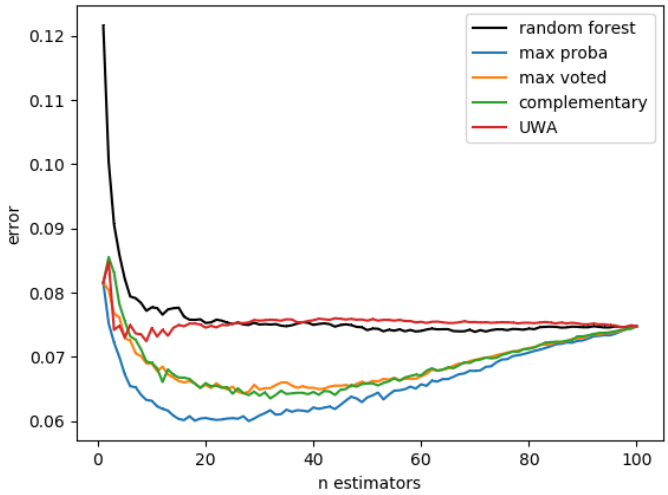

Figura 0-22: Precisión Phishing. Profundidad 5

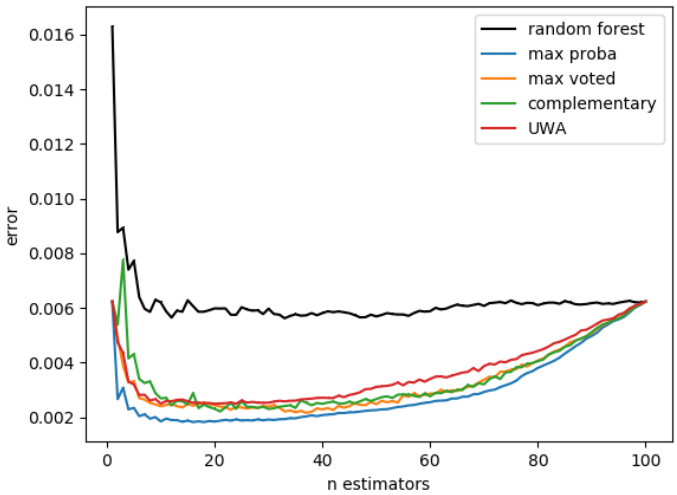

Figura 0-23: Precisión Phishing. Profundidad 7

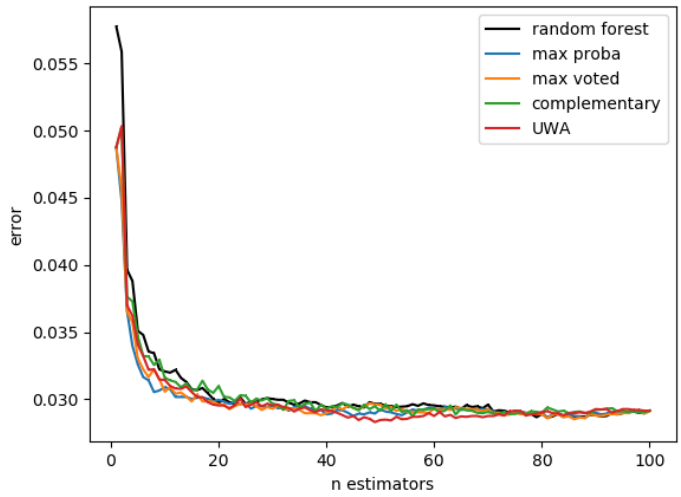

Figura 0-24: Precisión Phishing. Profundidad sin límite

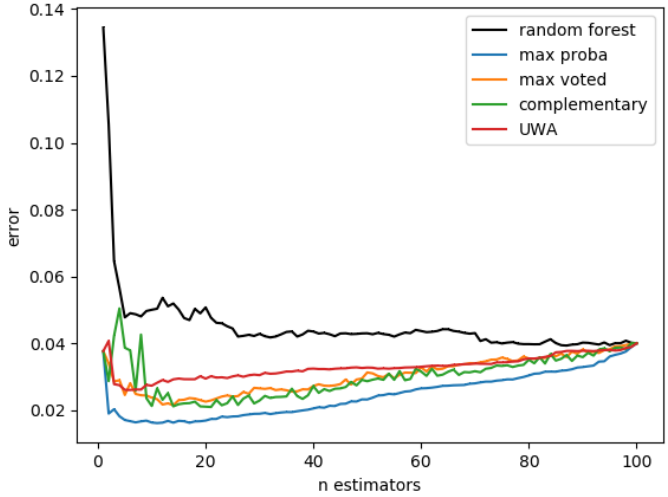

Figura 0-25: Precisión Sking segmentation. Profundidad 3

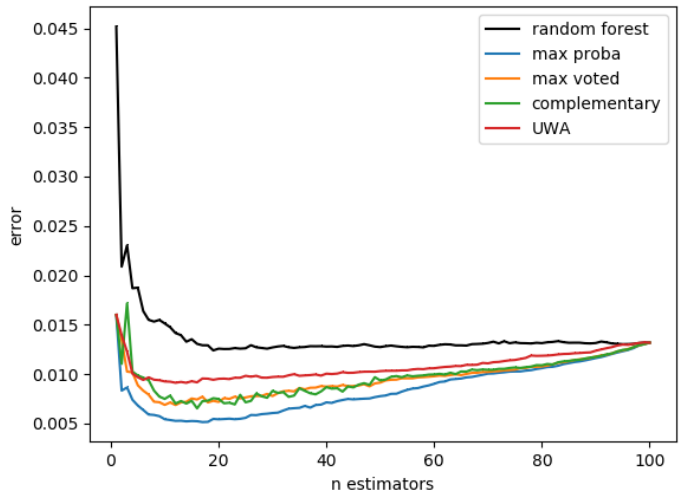

Figura 0-26: Precisión Sking segmentation. Profundidad 5

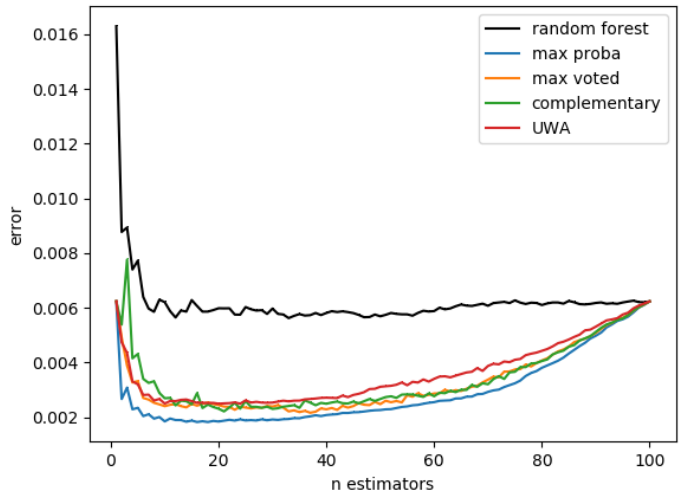

Figura 0-27: Precisión Sking segmentation. Profundidad 7

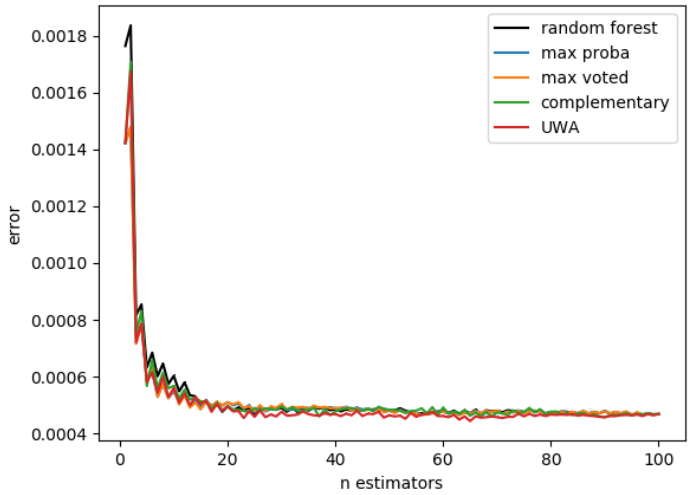

Figura 0-28: Precisión Sking segmentation. Profundidad sin límite

Pruebas de tasa de poda óptima

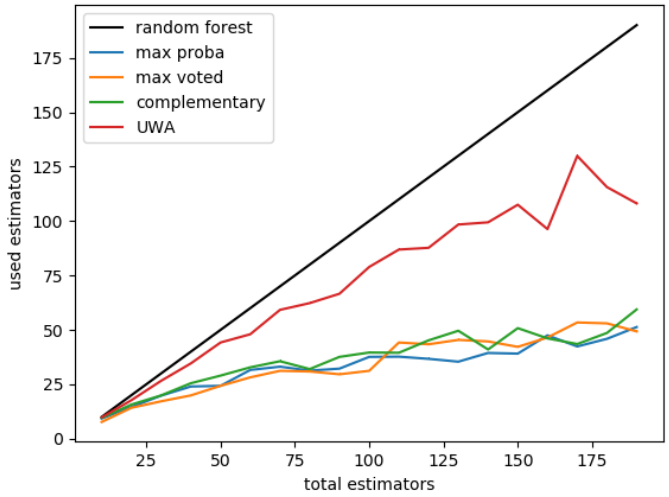

Figura 0-29: Tasa de poda Digits. Profundidad 3

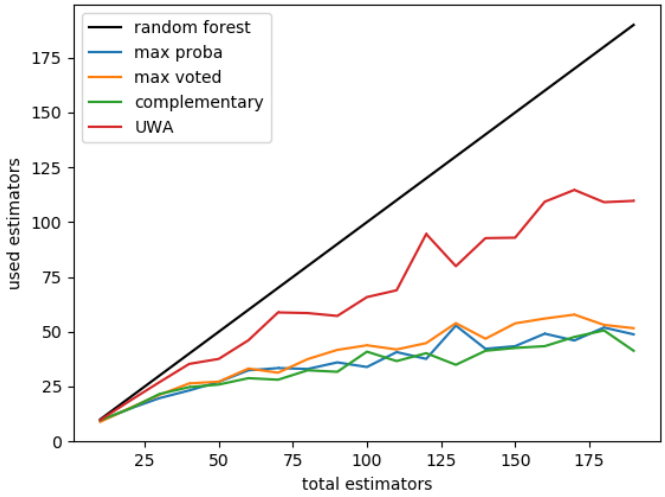

Figura 0-30: Tasa de poda Digits. Profundidad 5

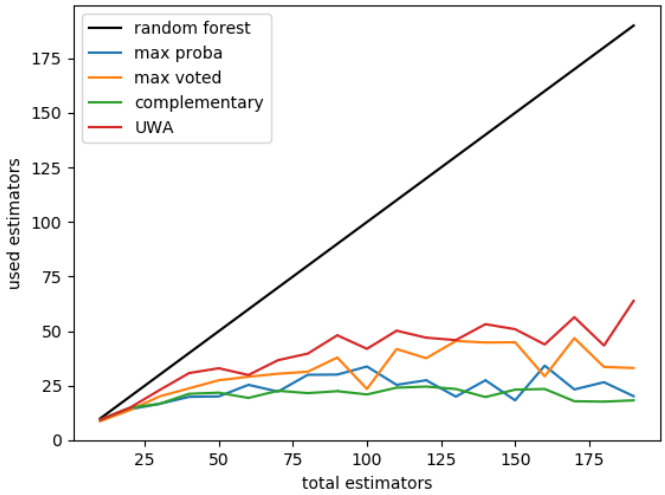

Figura 0-31: Tasa de poda Digits. Profundidad 7

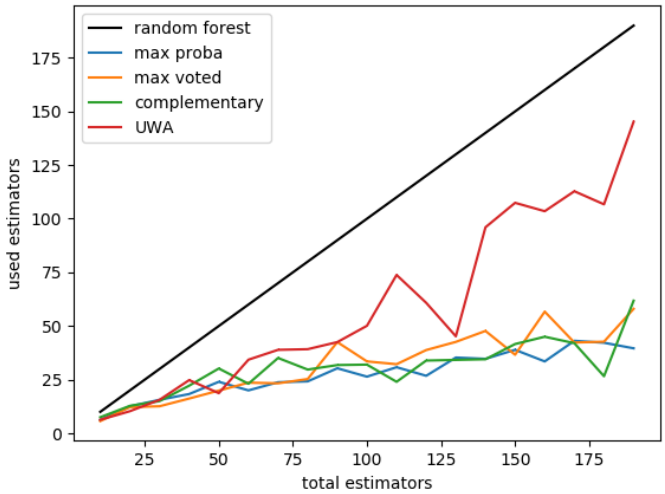

Figura 0-32: Tasa de poda Phishing. Profundidad 3

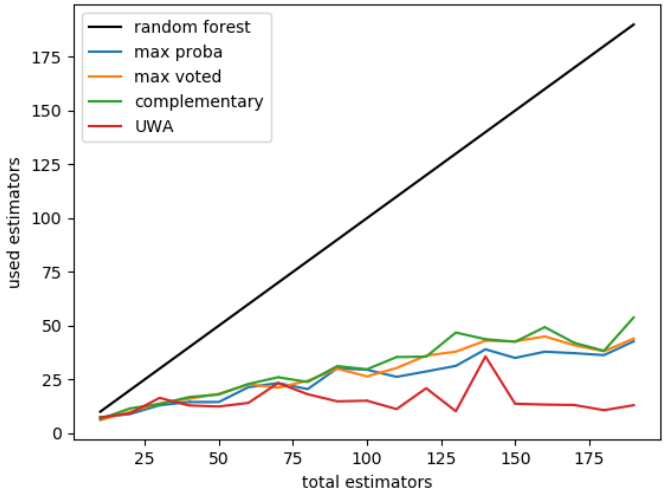

Figura 0-33: Tasa de poda Phishing. Profundidad 5

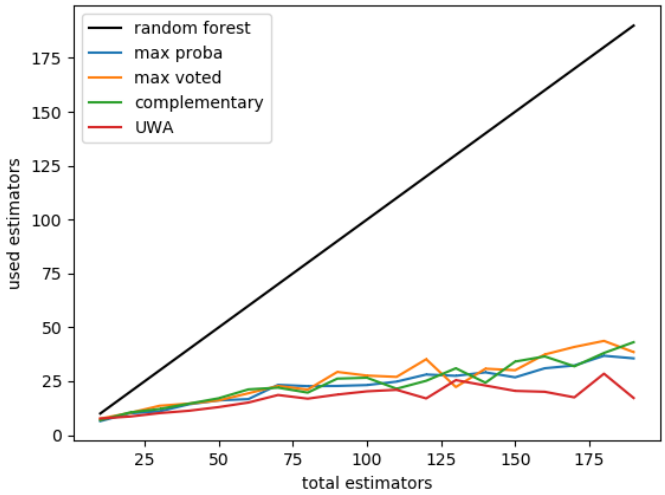

Figura 0-34: Tasa de poda Phishing. Profundidad 7

Pruebas de coste de entrenamiento

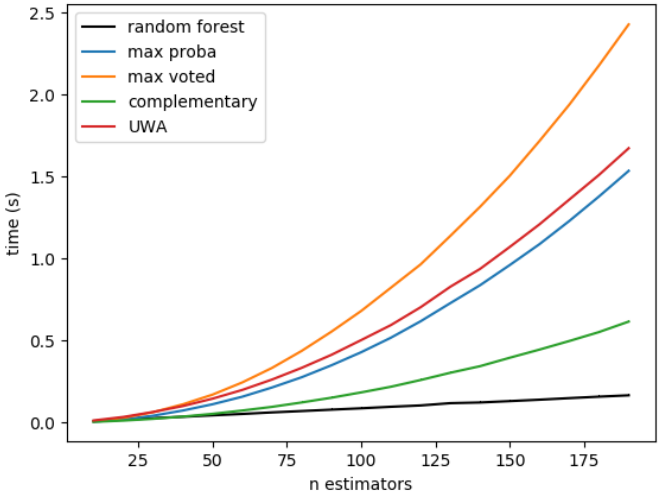

Figura 0-35: Coste entrenamiento Iris. Profundidad 3

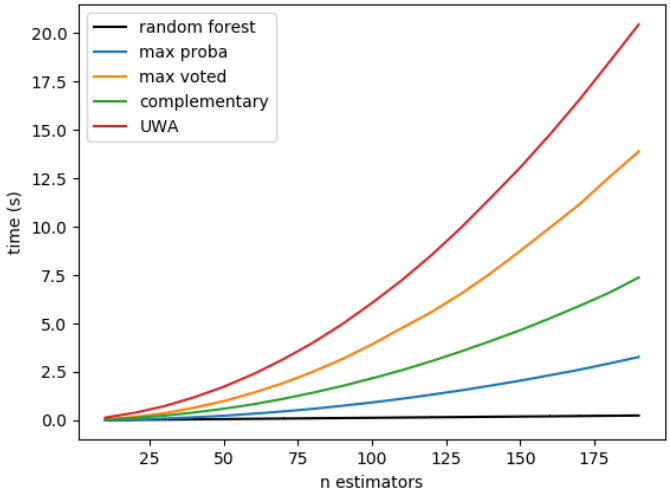

Figura 0-36: Coste entrenamiento Digits. Profundidad 3